HoGent

BEDRIJF
EN
ORGANISATIE

POD II - Deel II: Bomen en Grafen

Jens Buysse (Jens.Buysse@hogent.be)
Harm De Weirdt (Harm.DeWeirdt@hogent.be)
Wim Goedertier (Wim.Goedertier@hogent.be)
Stijn Lievens (Stijn.Lievens@hogent.be)
Lieven Smits (Lieven.Smits@hogent.be)

Deel II: Bomen en Grafen

Inhoud

Motivatie

Er zijn veel situaties waarin informatie geordend is volgens één of andere hiërarchische structuur. Denken we bv. maar aan

- bestandsystemen
- de structuur van XML-bestanden
- familiestambomen
- organisatorische structuren
- ...

De abstractie die zulke situaties modelleert is een *boom*, een fundamenteel begrip in de informatica.

Definitie

We geven een *recursieve* definitie.

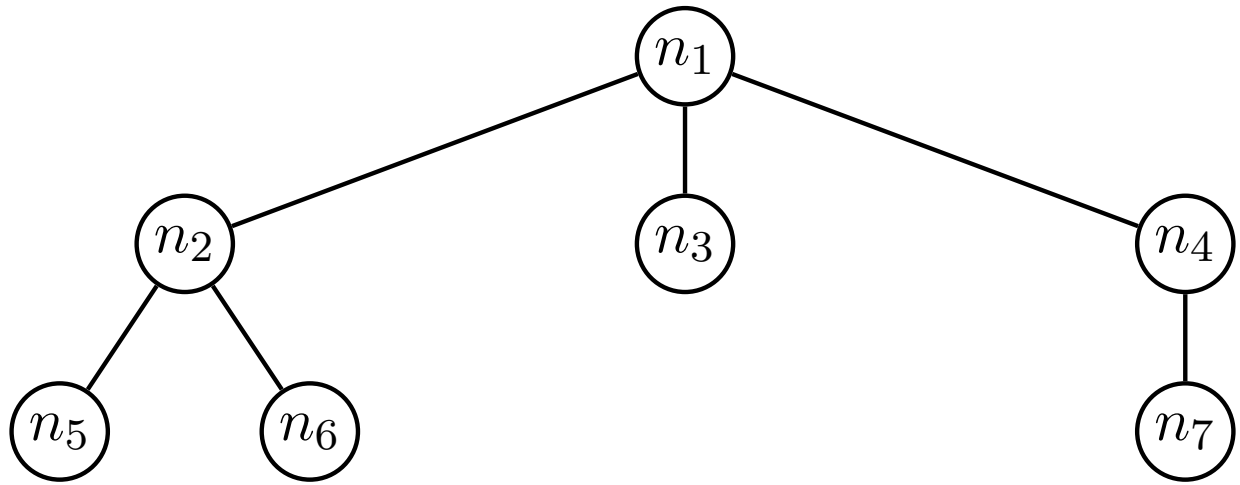
Definitie

Een GEWORTELDE BOOM T *is een verzameling van* TOPPEN *die aan de volgende eigenschappen voldoet:*

1. *Er is één speciale top* t *die de* WORTEL *van de boom wordt genoemd.*
2. *De andere toppen zijn verdeeld in* $m \geq 0$ *disjuncte verzamelingen* $T_1, \ldots, T_m$ *die op hun beurt elk weer een* GEWORTELDE BOOM *zijn.*

Opmerking:
Één enkele top is ook steeds een boom aangezien het toegelaten is dat het aantal verzamelingen m nul is. Dit is het eindgeval van de recursie

Grafische voorstelling



Voor deze boom is de wortel n_1. We hebben verder:

$$T_1 = \{n_2, n_5, n_6\}, \quad T_2 = \{n_3\} \quad \text{en} \quad T_3 = \{n_4, n_7\}.$$

Begrippen

- DEELBOOM
- KINDEREN: wortels van de deelbomen; OUDER
- BROERS: kinderen van dezelfde ouder
- AFSTAMMELING en VOOROUDER
- GRAAD: aantal kinderen van een top; GRAAD van een boom: maximum graad van zijn toppen
- BLAD: top zonder kinderen (versus INTERNE top)

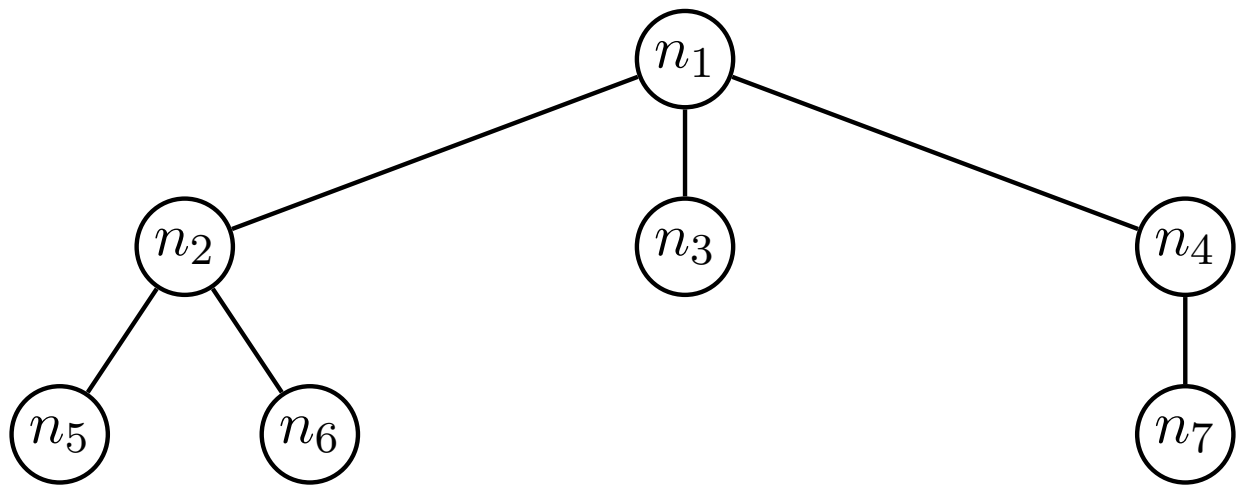
Diepte en hoogte

- De DIEPTE van een top n in een boom T.
  1. Diepte van de wortel is nul;
  2. Diepte van elke andere top is één meer dan de diepte van zijn ouder.
- De DIEPTE van de *boom* is de maximale diepte van zijn toppen.
- De HOOGTE van een top is de diepte van de deelboom met die top als wortel.
- De HOOGTE van een boom is de hoogte van zijn wortel (is dus ook de diepte van de boom!)

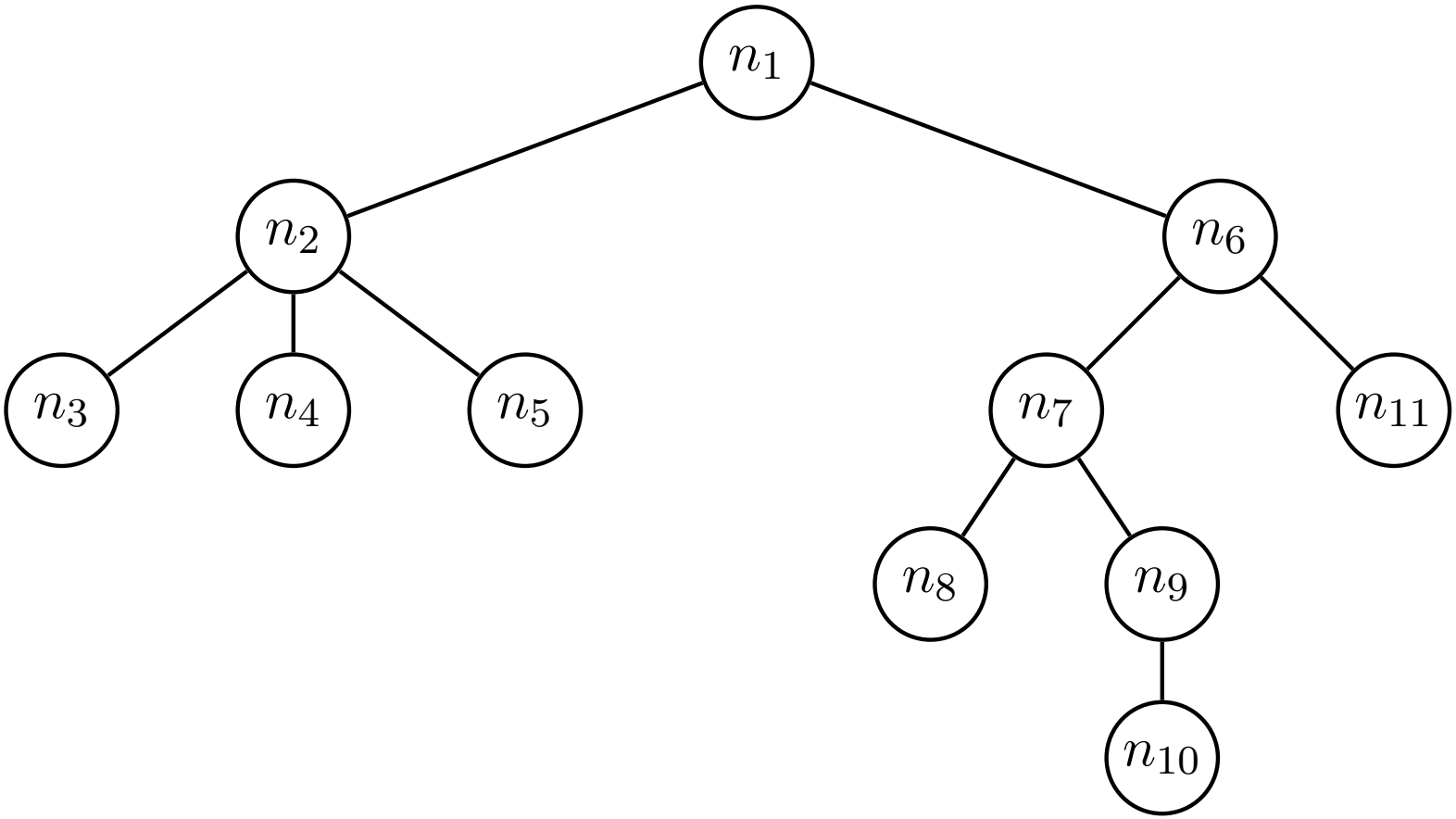
Samengevat: diepte is 'afstand' tot wortel; hoogte is 'afstand' tot verste blad.

Diepte en hoogte: voorbeeld

Bepaal de diepte en hoogte van alle toppen in de voorbeeldboom (die hieronder wordt herhaald).



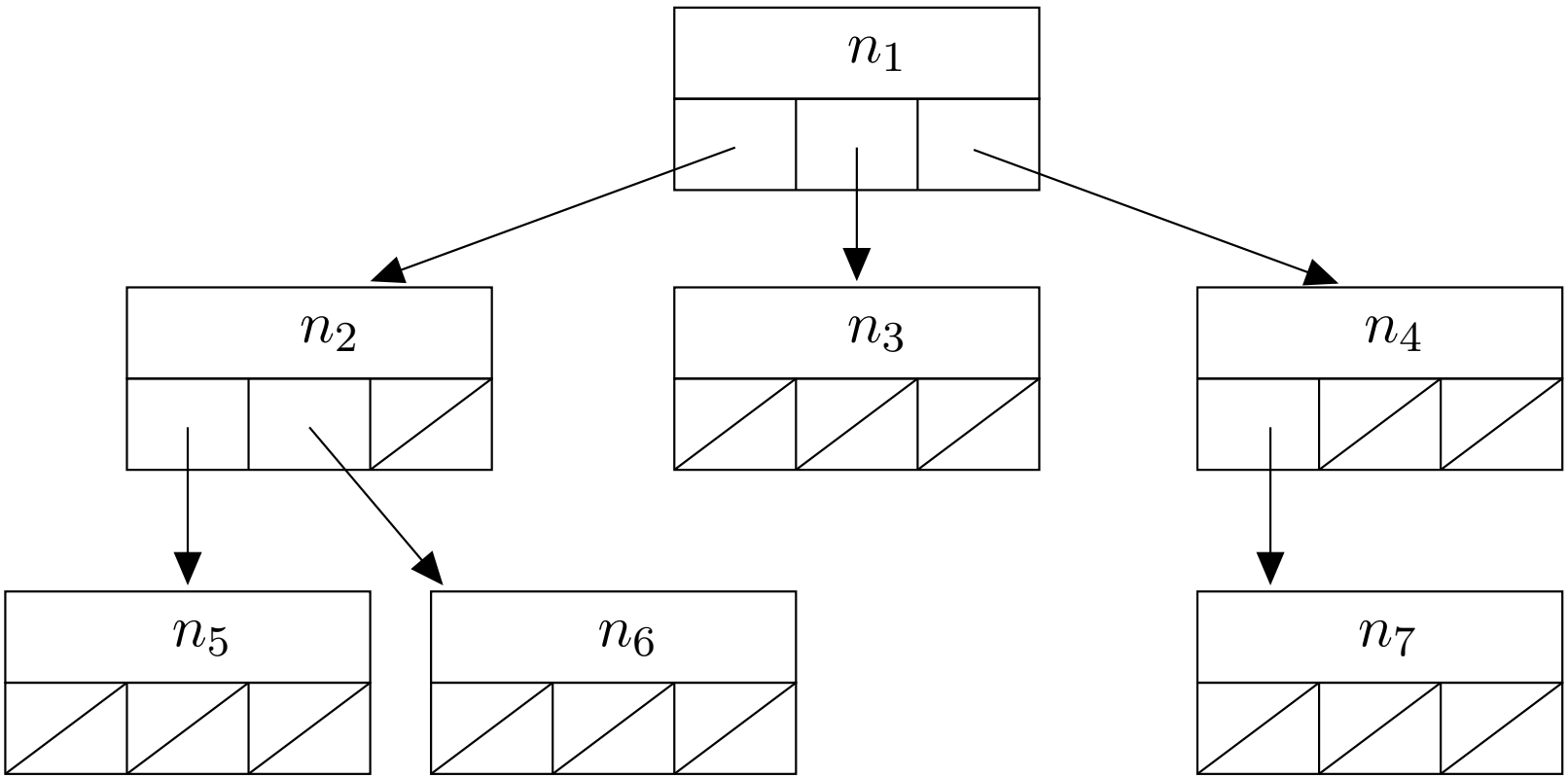
Oefeningen



Oefeningen

1. Bekijk de boom op vorige slide.
   - 1.1 Geef de wortel van de boom.
   - 1.2 Geef de verzamelingen T_1 t.e.m. T_m volgens Definitie 1.
   - 1.3 Geef de kinderen van elke top in de boom.
   - 1.4 Geef de graad van elke top in de boom.
   - 1.5 Geef de graad van de boom T.
   - 1.6 Welke toppen zijn broers van elkaar?
   - 1.7 Geef de bladeren van de boom.
   - 1.8 Geef de afstammelingen van n_6.
   - 1.9 Geef de voorouders van n_{10}.
   - 1.10 Maak een tabel waarin voor elke top zijn hoogte en diepte wordt gegeven.

Array-van-kinderen voorstelling



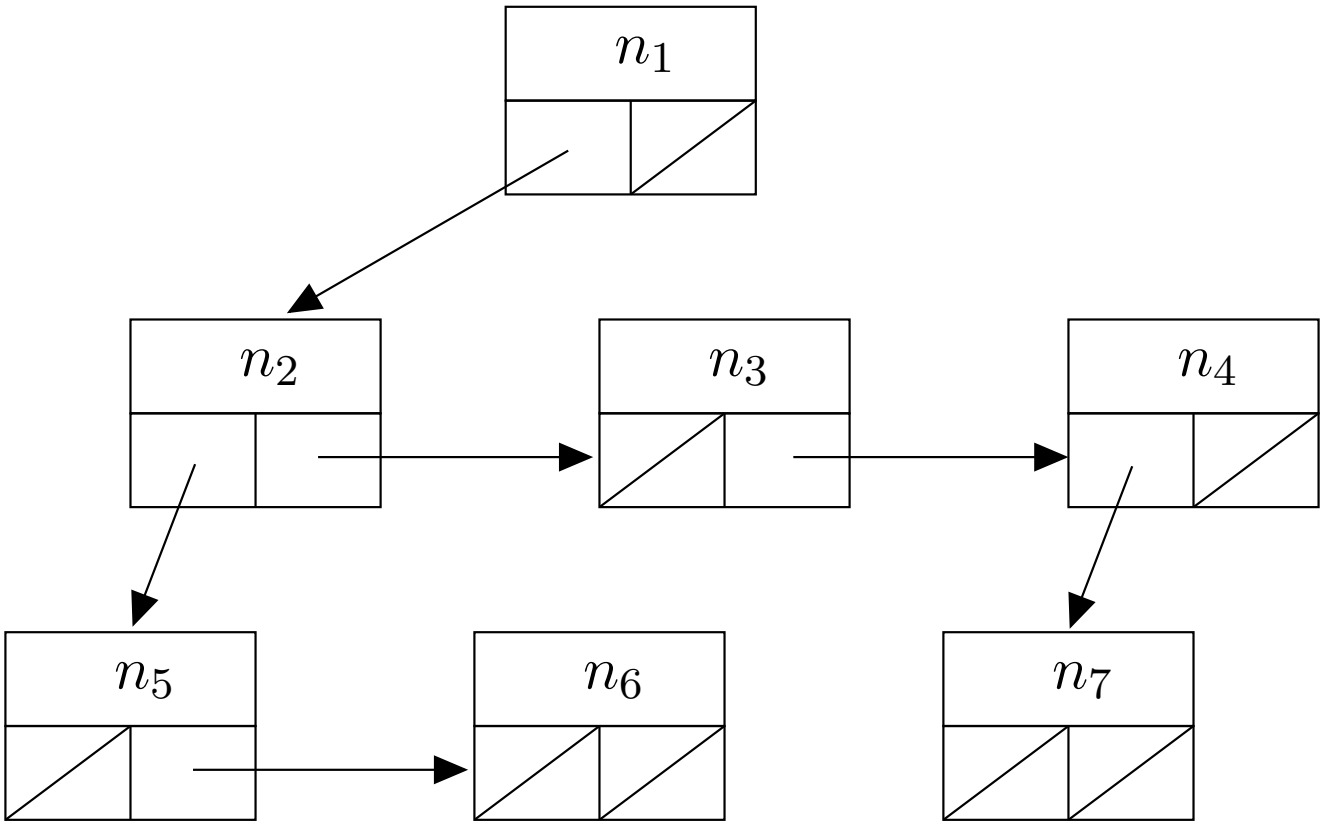
Datastructuur Top met een array van referenties naar kinderen. De grootte van de array is graad van de boom.

Geheugengebruik

- Aantal referenties: $n \times k$
- Aantal gebruikte referenties: $n - 1$ (waarom?)
- Verhouding is dus slechts

$$\frac{n-1}{nk} \approx \frac{1}{k}$$

Eerste-kind-volgende-broer voorstelling



Elke `Top` heeft juist twee referenties: een referentie naar zijn eerste kind, en een referentie naar zijn volgende broer.

Geheugengebruik

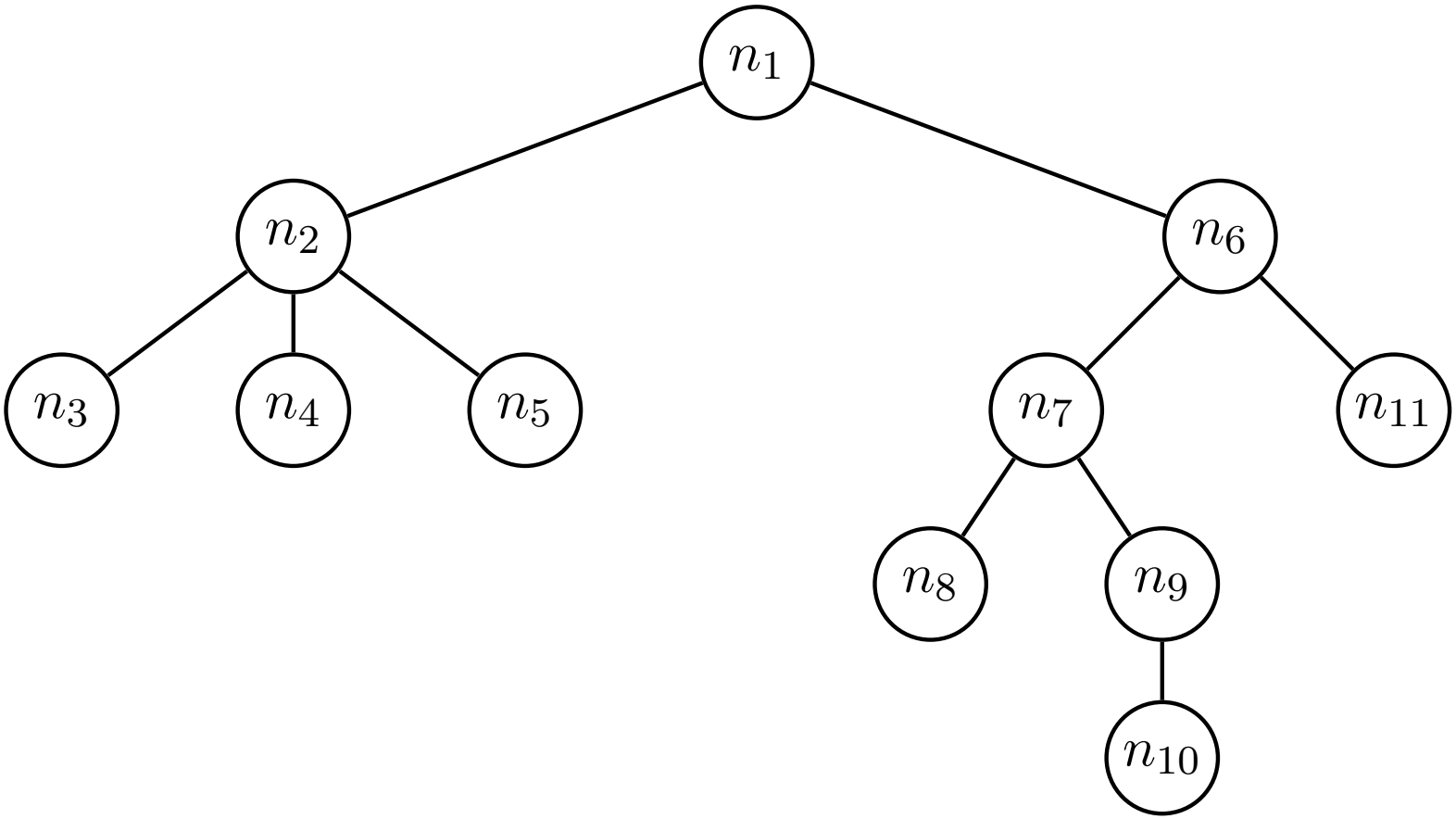
- Aantal referenties: $2n$ (onafhankelijk van k)
- Aantal gebruikte referenties: $n - 1$ (waarom?)
- Verhouding wordt nu

$$\frac{n-1}{2n} \approx \frac{1}{2},$$

  en is dus onafhankelijk van de graad van de boom.

Nadeel: bepalen/aanspreken van bv. derde kind wordt iets moeilijker.

Oefeningen



Oefeningen

Bekijk de boom op vorige slide (of Figuur 4.2 in de cursus).

1. Bereken hoeveel null-referenties er zullen zijn bij array-van-kinderen voorstelling van de boom. Wat is de voorhouding van het aantal effectief gebruikte referenties tot het aantal voorziene referenties?
2. Teken de array-van-kinderen voorstelling van de boom.
3. Bereken hoeveel null-referenties er zullen zijn bij eerste-kind-volgende-broer voorstelling van de boom. Wat is de verhouding van het aantal effectief gebruikte referenties tot het aantal voorziene referenties?
4. Wat is de voorhouding van het aantal effectief gebruikte referenties tot het aantal voorziene referenties voor een willekeurige gewortelde boom van graad k met n toppen.
5. Teken de eerste-kind-volgende-broer voorstelling van de boom.

Alle toppen van een boom bezoeken

Om een boom te doorlopen in PREORDE gaat men als volgt tewerk:

1. Bezoek (eerst) de wortel van de boom.
2. Doorloop de deelbomen van de wortel *in preorde* (recursie!)

Om een boom te doorlopen in POSTORDE doet men het volgende:

1. Doorloop de deelbomen van de wortel *in postorde* (recursie!).
2. Bezoek de wortel van de boom (laatst)

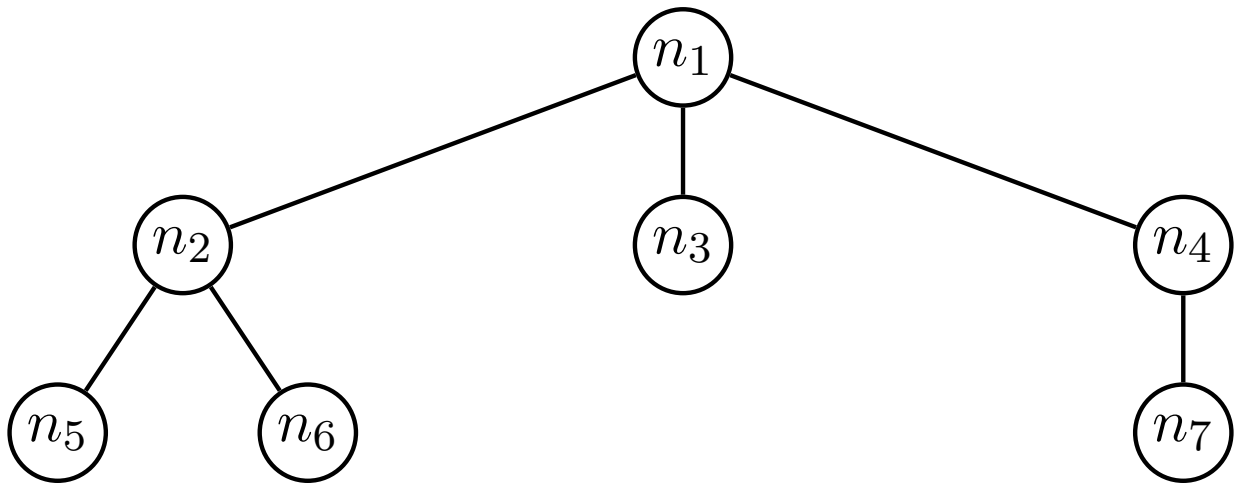
Eindgeval van de recursie: er zijn geen deelbomen (dus top is blad)

Preorde: pseudocode

Invoer Een boom T, en een visit functie.
Uitvoer De visit functie is aangeroepen voor elke top van de boom.

```
1: function PreOrde(T,visit)
2:     PreOrdeRecursief(T.wortel, visit)      ▷ start met de wortel
3: end function
4: function PreOrdeRecursief(v, visit)
5:     visit(v)
6:     for all w ∈ kinderen(v) do  ▷ implementatie-onafhankelijk
7:         PreOrdeRecursief(w, visit)
8:     end for
9: end function
```

Preorde: printen van toppen



Eenvoudige berekeningen

We wensen het aantal toppen van een boom te berekenen.
We weten:

$$\#(T) = 1 + \#(T_1) + \#(T_2) + \cdots + \#(T_m).$$

Dit kan nu eenvoudig vertaald worden in (pseudo)code.

Pseudocode aantal toppen

Invoer Een boom T
Uitvoer Het aantal toppen van de boom.

```
1:  function AANTAL(T)
2:      return AantalRecursief(T.wortel)
3:  end function
4:  function AANTALRECURSIEF(v)
5:      n ← 1                                   ▷ Top zelf niet vergeten
6:      for all w ∈ kinderen(v) do
7:          n ← n + AantalRecursief(w)
8:      end for
9:      return n
10: end function
```

Oefeningen

1. Geef de volgorde waarin de toppen worden bezocht wanneer de boom in Figuur 4.2 respectievelijk in preorde en postorde wordt doorlopen.
2. Geef code analoog aan Algoritme 4.1 om een gewortelde boom in postorde te doorlopen.
3. Geef code analoog aan Algoritme 4.2 om de hoogte van een gewortelde boom te berekenen. Baseer je op formule (4.2) uit de cursus.

Definitie

Definitie

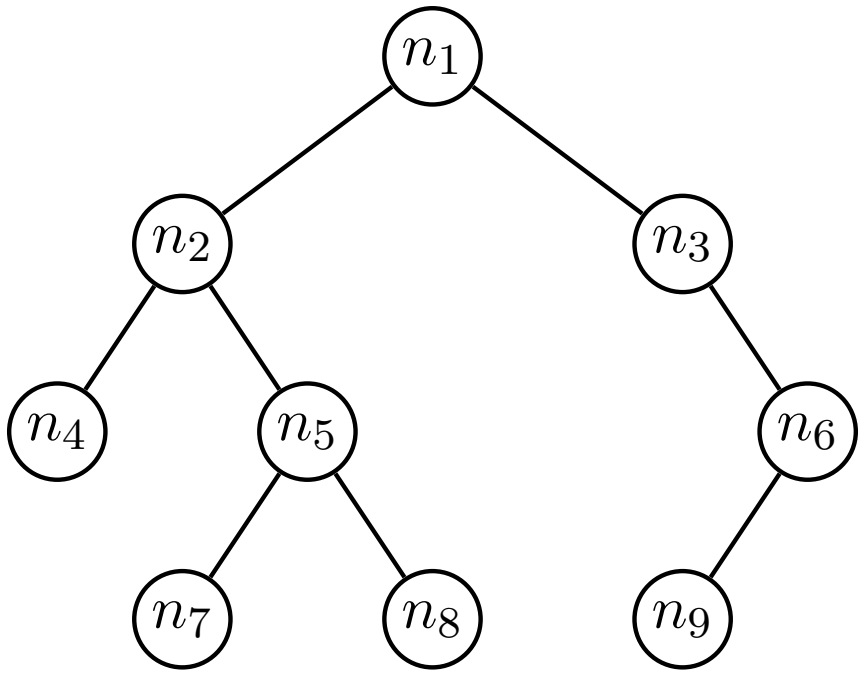
Een BINAIRE BOOM *is een verzameling toppen die*

1. *ofwel leeg is,*
2. *ofwel bestaat uit een* ***wortel*** *en twee disjuncte verzamelingen T_l en T_r, die op hun beurt ook een* BINAIRE BOOM *zijn. We noemen T_l en T_r respectievelijk de* ***linker-*** *en* ***rechterdeelboom*** *van de wortel.*

Opmerkingen:

- Binaire boom kan leeg zijn, "gewone" gewortelde boom niet;
- Bij binaire bomen zijn deelbomen geordend: linker- en rechterdeelboom.

Voorbeeld



Eigenschappen van binaire bomen

Stelling

In een binaire boom is het aantal toppen met diepte k hoogstens 2^k.

Bewijs.

Door inductie op de diepte k. □

Eigenschappen van binaire bomen

Als we de diepte van een binaire boom kennen dan kunnen we grenzen opstellen voor het aantal toppen in deze boom.

Stelling

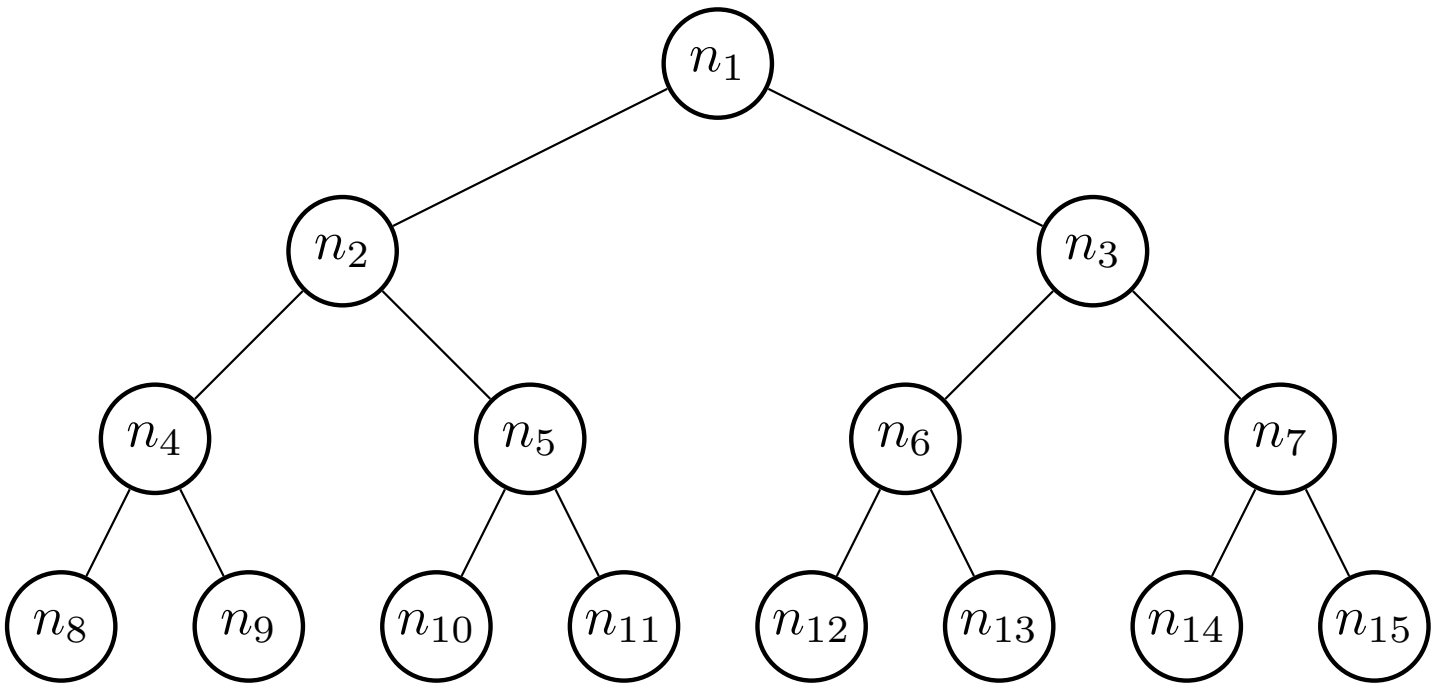
Voor een (niet-lege) binaire boom T met diepte $d \geq 0$ geldt dat:

$$d + 1 \leq \#(T) \leq 2^{d+1} - 1. \tag{1}$$

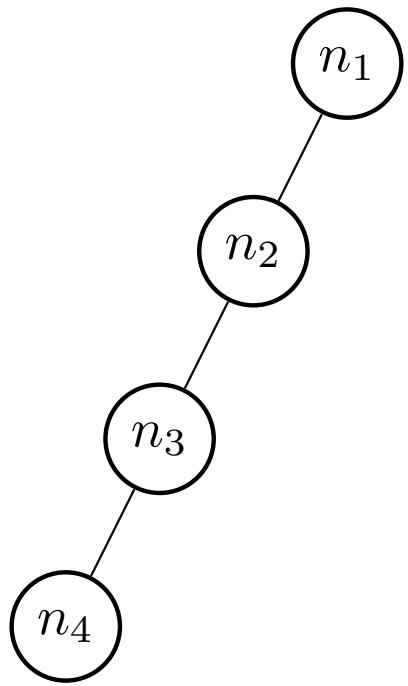
Bewijs.

Zie bord! □

Illustratie

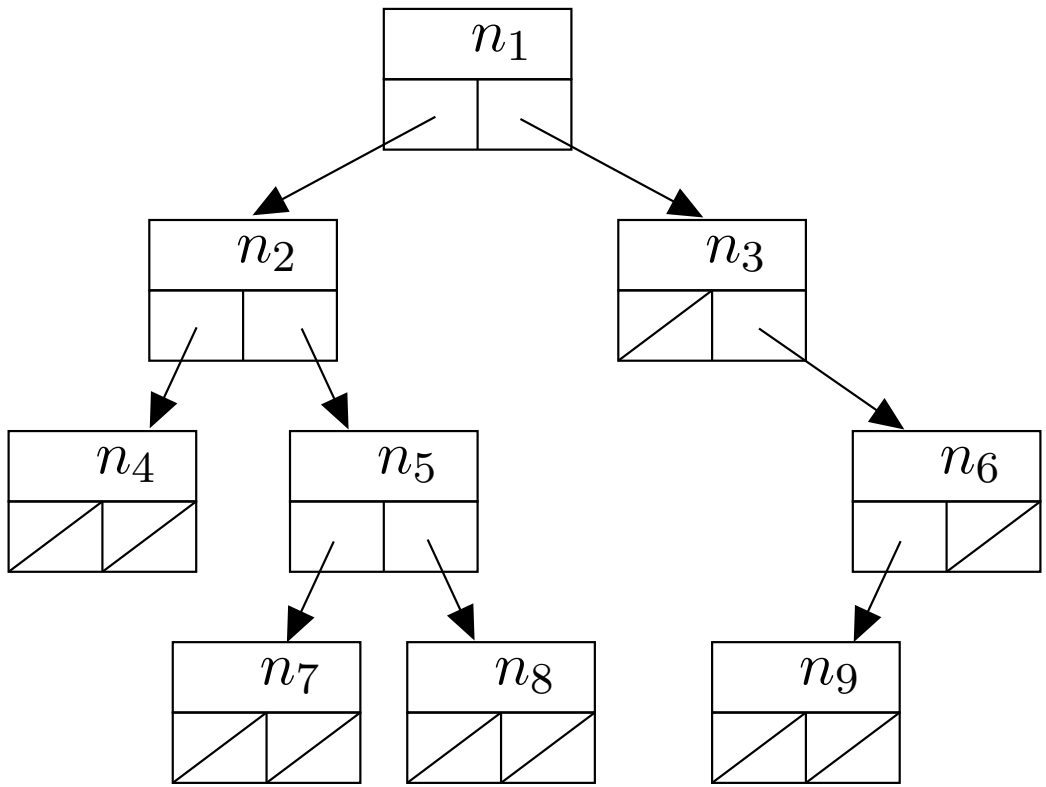


Illustratie



Voorstelling van een binaire boom

We gebruiken een datastructuur `Top` met twee referentievelden links en rechts.
De boom zelf is eigenlijk een referentie naar zijn wortel.



Alle toppen bezoeken

Om een binaire boom in PREORDE te doorlopen gaan we dus als volgt tewerk:

1. Bezoek de wortel van boom.
2. Als de linkerdeelboom niet leeg is, doorloop de linkerdeelboom dan *recursief in preorde*.
3. Als de rechterdeelboom niet leeg is, doorloop de rechterdeelboom dan *recursief in preorde*.

POSTORDE idem maar bezoek aan wortel laatst.
Derde mogelijkheid: om een binaire boom in INORDE te doorlopen gaan we als volgt tewerk:

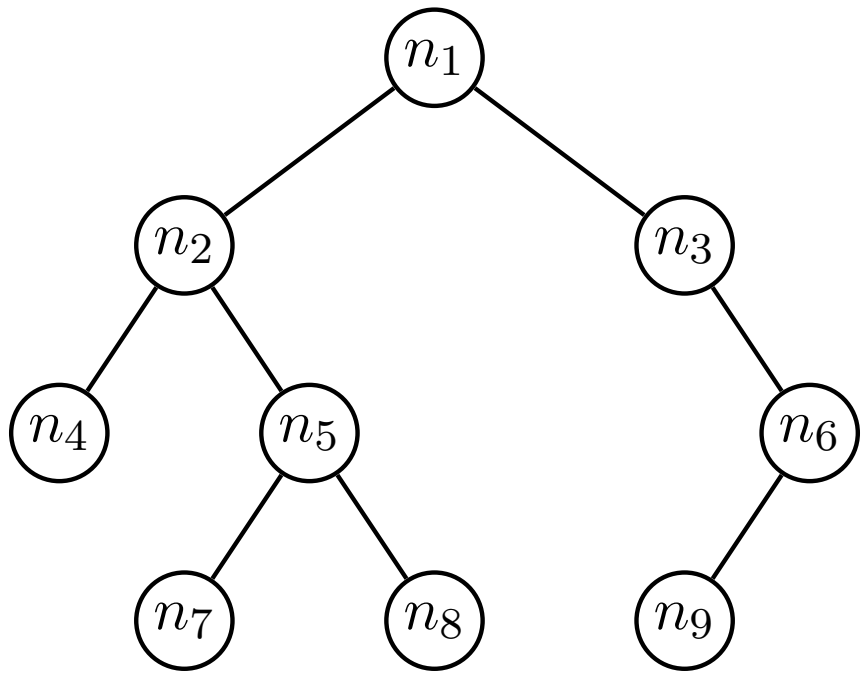
1. Als de linkerdeelboom niet leeg is, doorloop de linkerdeelboom dan *recursief in inorde*.
2. Bezoek de wortel van boom.
3. Als de rechterdeelboom niet leeg is, doorloop de rechterdeelboom dan *recursief in inorde*.

Invoer Een binaire boom T en een visit functie.
Uitvoer De visit functie is aangeroepen voor elke top van T.

```
1: function InOrde(T,visit)
2:     if T ≠ ∅ then                    ▷ controleer dat boom niet leeg is
3:         InOrdeRecursief(T.wortel, visit)   ▷ start met de wortel
4:     end if
5: end function
6: function InOrdeRecursief(v, visit)
7:     if v.links ≠ ∅ then
8:         InOrdeRecursief(v.links, visit)
9:     end if
10:    visit(v)          ▷ visit aanroepen tussen recursieve oproepen
11:    if v.rechts ≠ ∅ then
12:        InOrdeRecursief(v.rechts, visit)
13:    end if
14: end function
```

Voorbeeld

Druk de labels van onderstaande boom af in preorde, inorde, en postorde.



Oefeningen

1.
   - Teken de binaire boom met labels 0 t.e.m. 9 waarvoor de inorde sequentie

     $$9, 3, 1, 0, 4, 2, 7, 6, 8, 5$$

     is terwijl de postorde sequentie

     $$9, 1, 4, 0, 3, 6, 7, 5, 8, 2$$

     is.
   - Doe nu hetzelfde voor de volgende sequenties, of leg uit waarom zo'n binaire boom niet bestaat:

     inorde: $9, 3, 1, 0, 4, 2, 7, 6, 8, 5$

     en

     postorde: $9, 1, 4, 0, 3, 6, 5, 7, 8, 2$

Collection

Vaak wensen we

1. gegevens toe te voegen aan de verzameling,
2. gegevens te verwijderen uit de verzameling, en
3. te controleren of een element aanwezig is in de verzameling.

Mogelijke datastructuren: (lineair gelinkte) lijsten en hash-tabellen, etc.

Een andere mogelijke datastructuur is de *binaire zoekboom*.

Totaal geordende verzameling

Een noodzakelijke voorwaarde om een binaire zoekboom te kunnen gebruiken is dat de labels een *totaal geordende verzameling* vormen.
Voor elke paar (mogelijke) labels x en y geldt dat

$$x < y \quad \text{of} \quad y < x \quad \text{of} \quad x = y.$$

Opmerking:
Vaak is er met het label andere data geassocieerd. Dit wordt in de theorie niet bekeken of vermeld.

Binaire zoekboom: definitie

Definitie

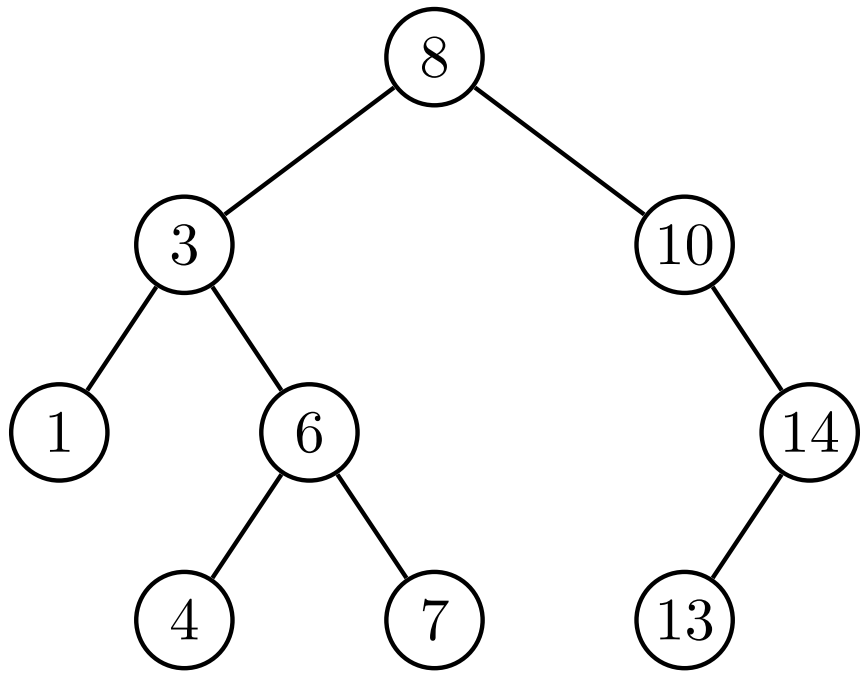
Een BINAIRE ZOEKBOOM *is een gelabelde binaire boom die aan een bijzondere voorwaarde, de binaire zoekboomeigenschap, voldoet.*

Definitie

De BINAIRE ZOEKBOOMEIGENSCHAP *is de volgende: voor elke top x van de binaire zoekboom geldt dat alle toppen in de linkerdeelboom van x een label hebben dat kleiner is dan het label van x, terwijl voor alle toppen in de rechterdeelboom van x geldt dat hun label groter is dan het label van x.*

Voorbeeld

Controleer dat onderstaande boom een geldige binaire zoekboom is.



Geef de labels van de boom in inorde. Wat merk je?

Opzoeken van een sleutel

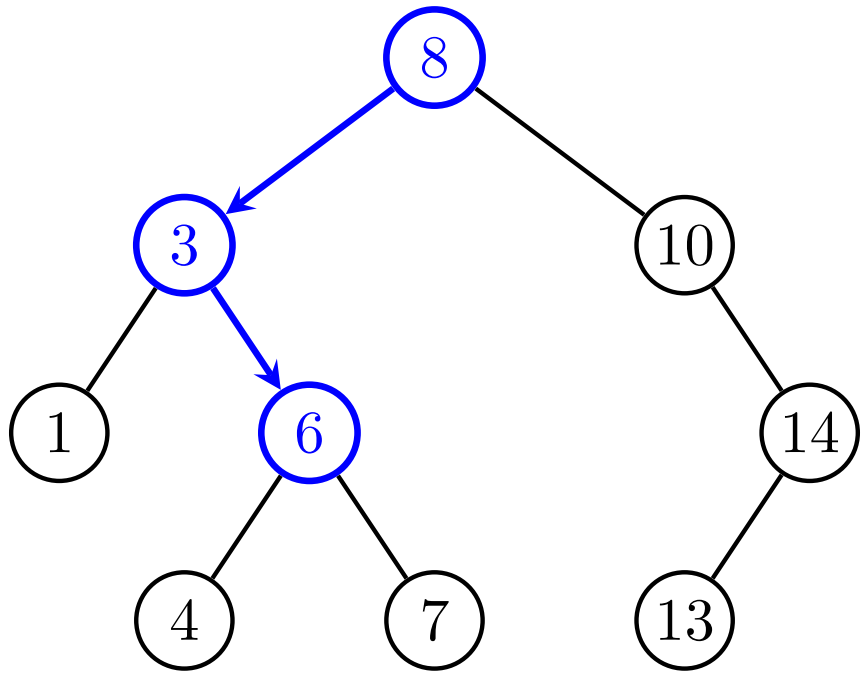
Om een sleutel op te zoeken in een binaire zoekboom maken we gebruik van de binaire zoekboomeigenschap om die (snel) te vinden.

1. Wanneer de boom leeg is, geef dan 'niet gevonden' terug.
2. Vergelijk x met de sleutel van de wortel.
   - 2.1 Wanneer x kleiner is dan dit label, zoek dan (recursief) in de linkerdeelboom.
   - 2.2 Wanneer x groter is dan dit label, zoek dan (recursief) in de rechterdeelboom.
   - 2.3 Geef de wortel van de boom terug (x werd gevonden).

Dit is dus een *recursieve* procedure.

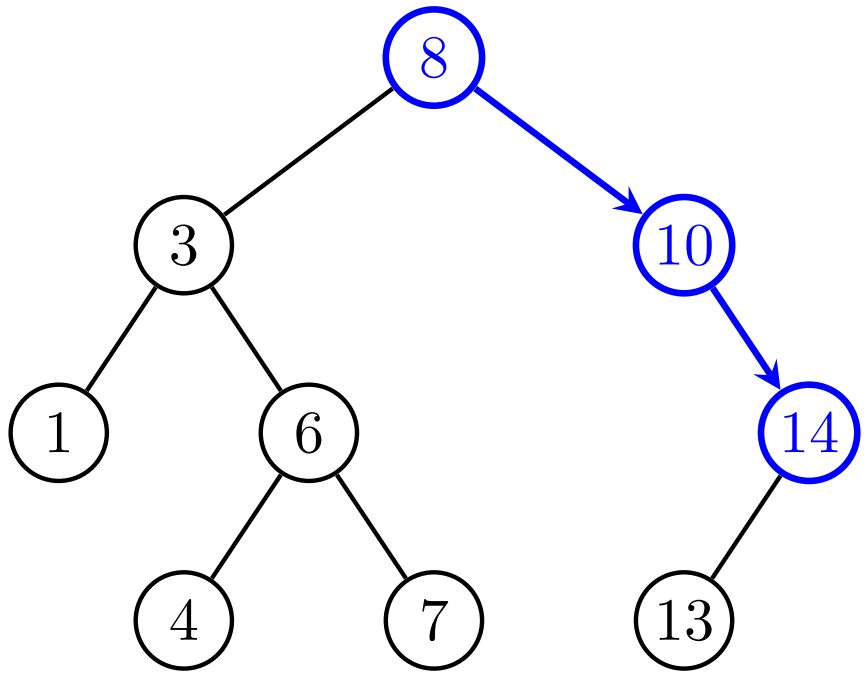
Voorbeeld

Zoek de sleutel 6 in de voorbeeldboom.



Voorbeeld

Zoek de sleutel 15 in de voorbeeldboom.



Opzoeken van kleinste/grootste element

Door de binaire zoekboomeigenschap weten we waar het kleinste element zich bevindt. Waar?

We vinden het kleinste element dus als volgt:

1. Wanneer de linkerdeelboom van de wortel leeg is, geef dan (de sleutel van) de wortel terug.
2. In het andere geval zoek je recursief naar het kleinste element van de linkerdeelboom.

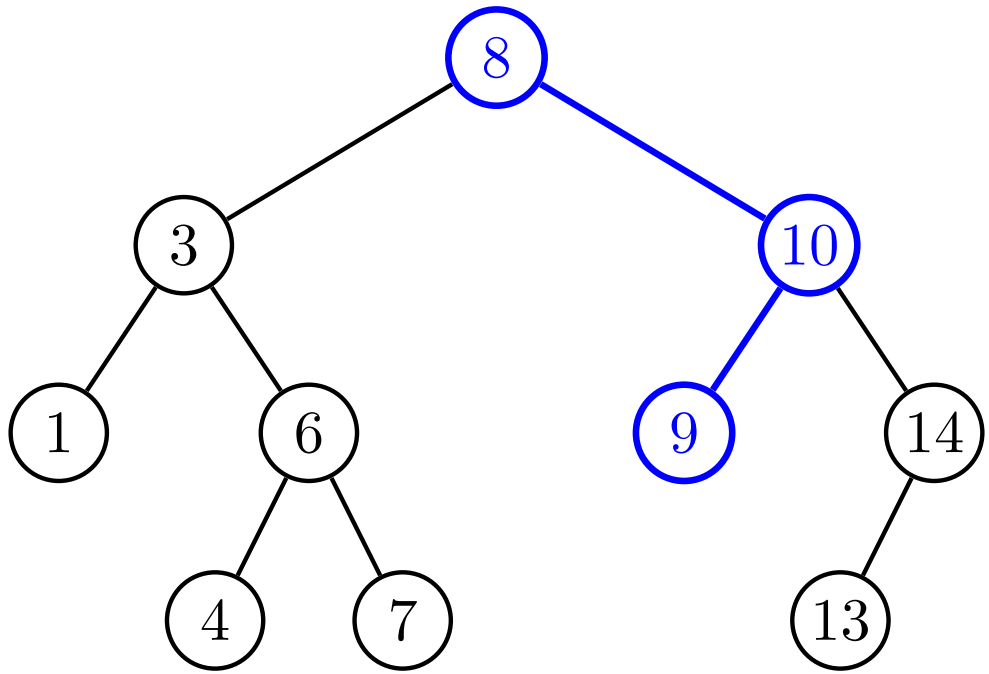
Toevoegen van een element

Na toevoegen van sleutel x moet nieuwe boom nog steeds binaire zoekboom zijn!
Toevoegen van een sleutel x aan een (niet-lege) binaire zoekboom:

1. Vergelijk x met het label van de wortel.
   - 1.1 Wanneer x kleiner is dan het label van de wortel, voeg dan x toe aan de linkerdeelboom wanneer die niet leeg is (recursie!). Wanneer de linkerdeelboom leeg is vervang dan de (`null`)-referentie naar de linkerdeelboom door de referentie naar een nieuwe top met x als label.
   - 1.2 Wanneer x groter is dan het label van de wortel, voeg dan x toe aan de rechterdeelboom wanneer die niet leeg is (recursie!). Wanneer de rechterdeelboom leeg is vervang dan de (`null`)-referentie naar de rechterdeelboom door de referentie naar een nieuwe top met x als label.
   - 1.3 Doe niets, want x behoort reeds tot de boom.

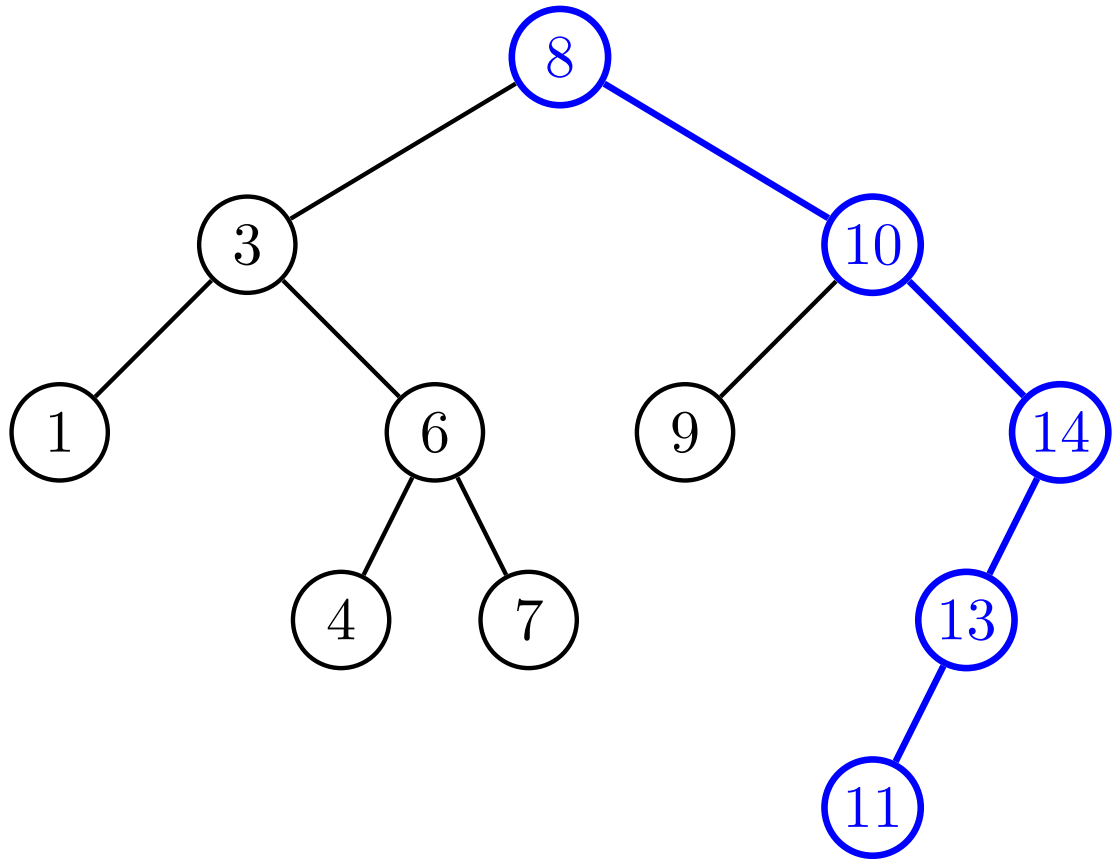
Voorbeeld

Toevoegen van sleutel 9 aan binaire zoekboom.



Voorbeeld

Toevoegen van sleutel 11 aan vorige binaire zoekboom.



Verwijderen van een sleutel

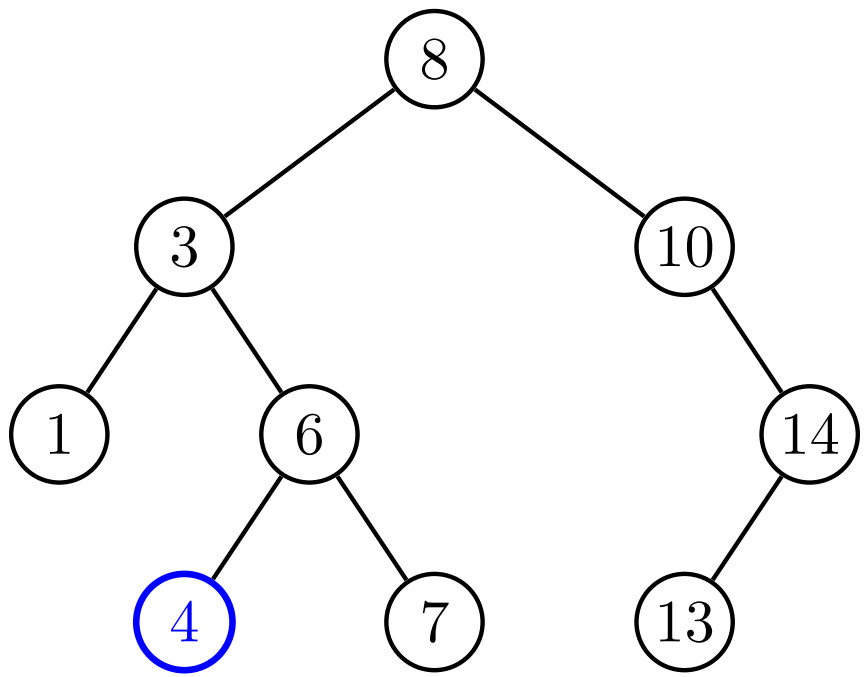
Verwijderen is de moeilijkste bewerking!

Het verwijderen van een sleutel x start met het opzoeken van deze sleutel in de boom. Er kunnen zich nu drie gevallen voordien:

1. De sleutel x bevindt zich in een blad.
2. De sleutel x bevindt zich in een top met één kind.
3. De sleutel x bevindt zich in een top met twee kinderen.

Verwijderen van een blad

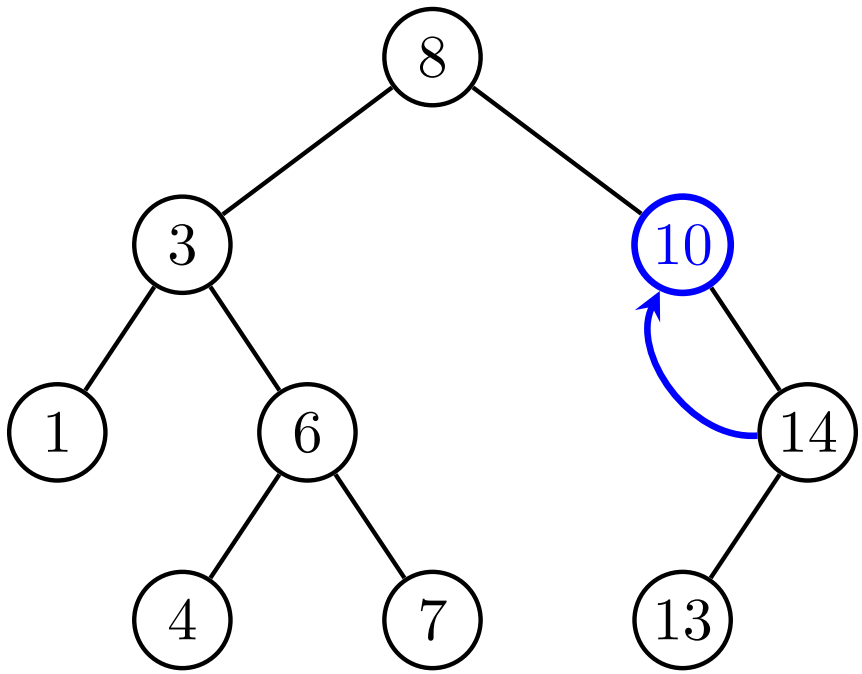
Laat eenvoudigweg dit blad weg. (Stel gepaste wijzer van ouder op `null`).
Verwijder sleutel 4 uit onderstaande boom.



Verwijderen top met één kind

Wanneer de top n die x bevat slechts één kind heeft, dan kunnen we n vervangen door dit ene kind. Anders gezegd: het nieuwe linker- of rechterkind van de ouder van n is zijn vroegere kleinkind. De binaire zoekboomeigenschap zal in dit geval nog steeds geldig zijn.

Verwijder 10 uit onderstaande boom



Verwijderen van een top met twee kinderen

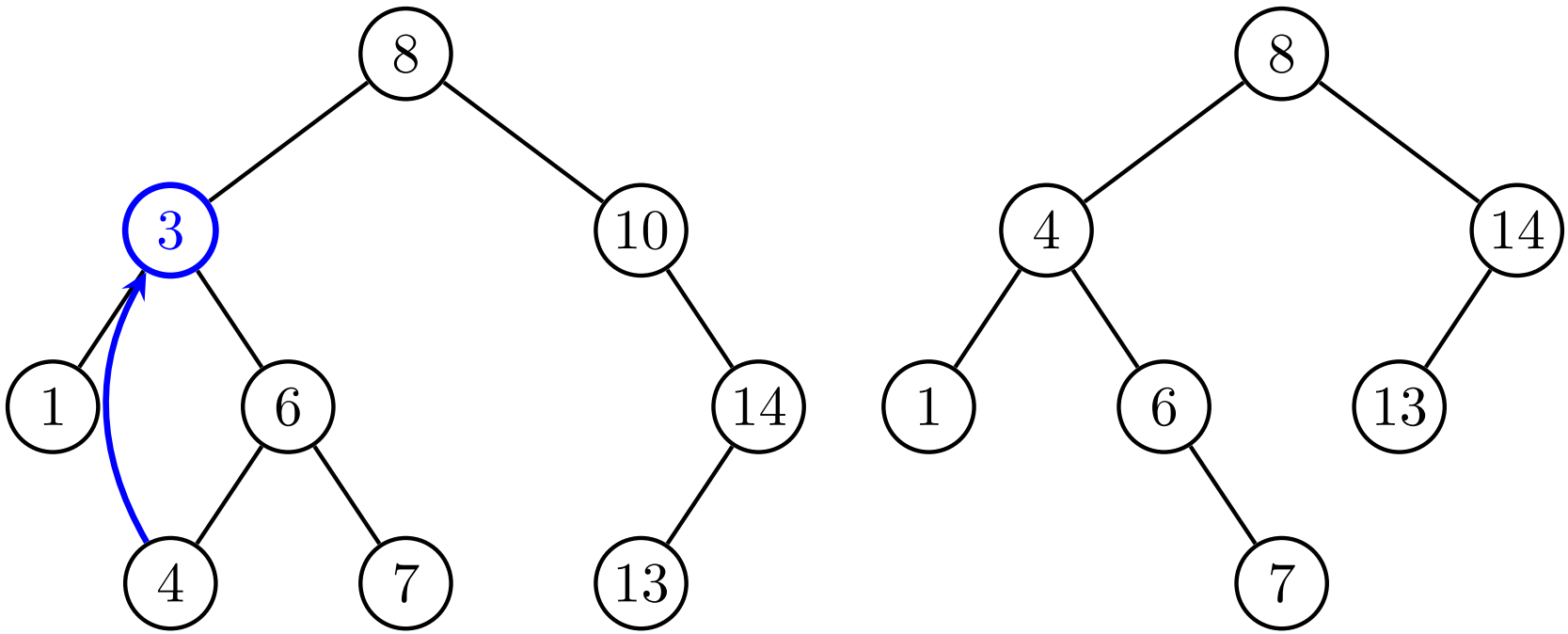
We herleiden dit probleem naar een eenvoudiger probleem:

Eerst vervangen we de inhoud van de top door zijn *opvolger*, i.e. de inhoud van de kleinste top in zijn rechterdeelboom.

Vervolgens verwijderen we deze kleinste top van de rechterdeelboom. (Waarom is dit eenvoudiger?!)

Voorbeeld

Verwijder de top met sleutel 3.



Tijdscomplexiteit

Al deze bewerkingen hebben een tijdscomplexiteit die lineair is in de diepte van de boom.

1. Voor 'perfect' gebalanceerde bomen: $\Theta(d) = \Theta(\lg(n))$.
2. In het slechtste geval: $\Theta(d) = \Theta(n)$
3. Gemiddelde geval (bij random toevoegen sleutels): $\Theta(d) = \Theta(\lg(n))$

Oefeningen

1. Geef de binaire zoekboom die opgebouwd wordt door de volgende sleutels

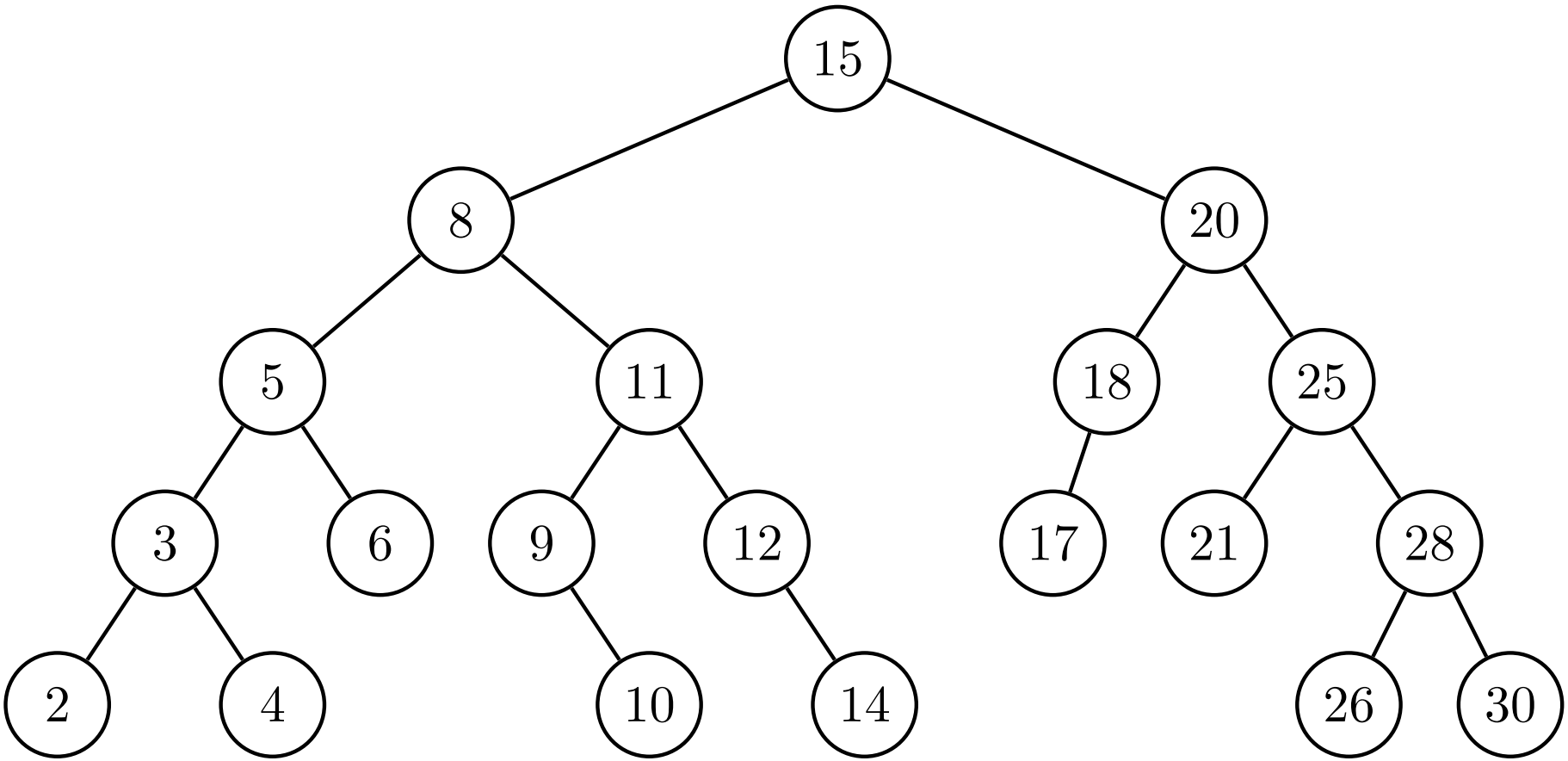
$$4, 7, 5, 8, 11, 3, 2, 9, 10, 6$$

   één voor één aan de zoekboom toe te voegen in de gegeven volgorde.
2. Veronderstel dat men een binaire zoekboom opbouwt door sleutels één voor één toe te voegen aan een initieel lege boom.

   2.1 Geef een rij van lengte 7 die een binaire zoekboom van minimale diepte oplevert.
   2.2 Geef een rij van lengte 15 die een binaire zoekboom van minimale diepte oplevert.
   2.3 Geef een rij van lengte 5 die een binaire zoekboom van maximale diepte oplevert. Wanneer krijg je een over het algemeen een slecht gebalanceerde binaire zoekboom?

Oefeningen

Dit is de binaire zoekboom voor de volgende oefening:



Oefeningen

3. Beschouw de binaire zoekboom in Figuur 4.15. Voer de volgende opdrachten één na één uit.
   - 3.1 Welke toppen worden er bezocht bij het zoeken naar de top 12?
   - 3.2 Welke toppen worden er bezocht bij het zoeken naar de top 27?
   - 3.3 Voeg een top met sleutelwaarde 23 toe aan de zoekboom. Teken de resulterende zoekboom.
   - 3.4 Voeg vervolgens een top met sleutelwaarde 22 toe aan de zoekboom. Teken de resulterende zoekboom.
   - 3.5 Verwijder de top met waarde 4 uit de zoekboom. Teken de resulterende zoekboom.
   - 3.6 Verwijder vervolgens de top met waarde 18 uit de zoekboom. Teken de resulterende zoekboom.
   - 3.7 Verwijder vervolgens de top met waarde 20 uit de zoekboom. Teken de resulterende zoekboom.

Prioriteitswachtrij

Een PRIORITEITSWACHTRIJ is een uitbreiding van de "gewone" FIFO wachtrij.
Elementen: sleutel en waarde. Hoe kleiner de sleutel, hoe groter de prioriteit.
Bij een prioriteitswachtrij kan men

1. het element met de kleinste sleutel opzoeken;
2. het element met de kleinste sleutel verwijderen;
3. een nieuw element toevoegen aan de wachtrij.

Opmerking:
In een prioriteitswachtrij kan *enkel* het element met de kleinste sleutel efficiënt bereikt worden. Dus invoegen is flexibel, verwijderen niet.

Binaire hoop

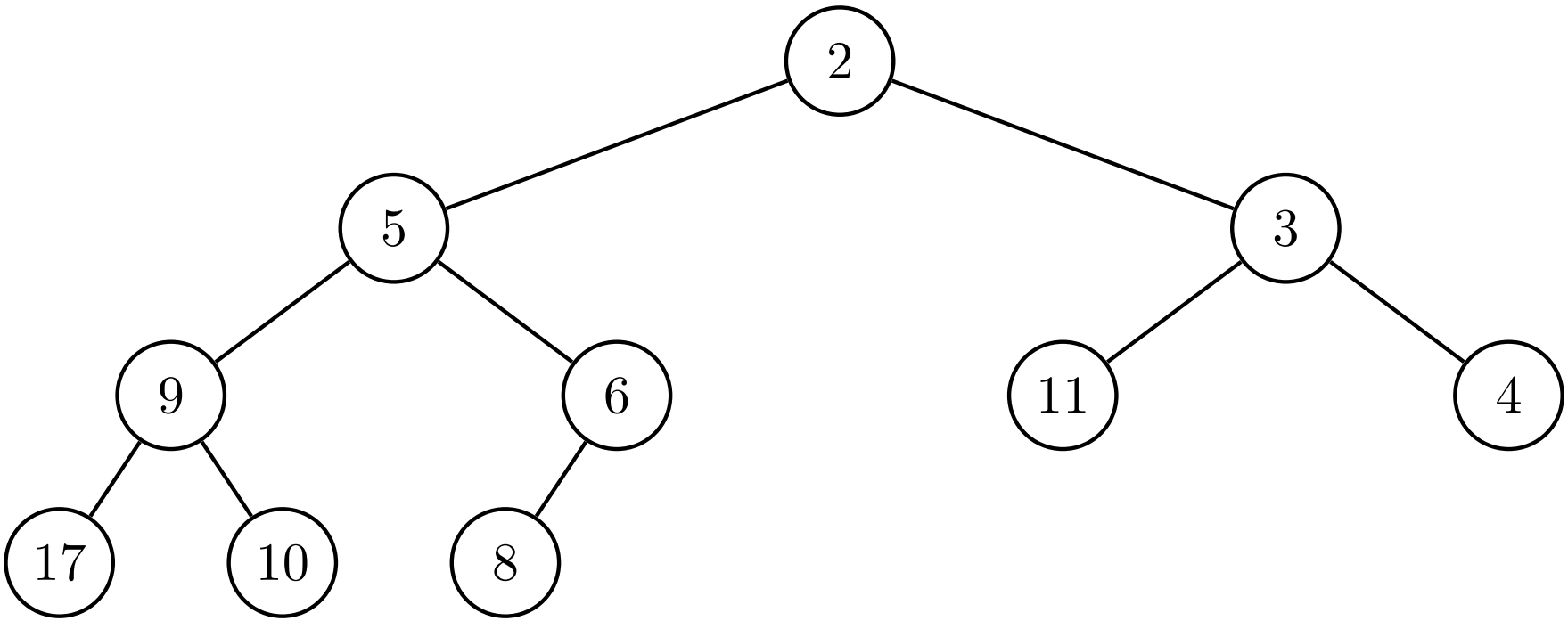
Definitie

Een COMPLETE BINAIRE BOOM *is een binaire boom van diepte d waarbij het aantal toppen met diepte $k < d$ maximaal (dus 2^k, zie Eigenschap 1) is. De toppen met diepte d komen voor van "links naar rechts".*

Definitie

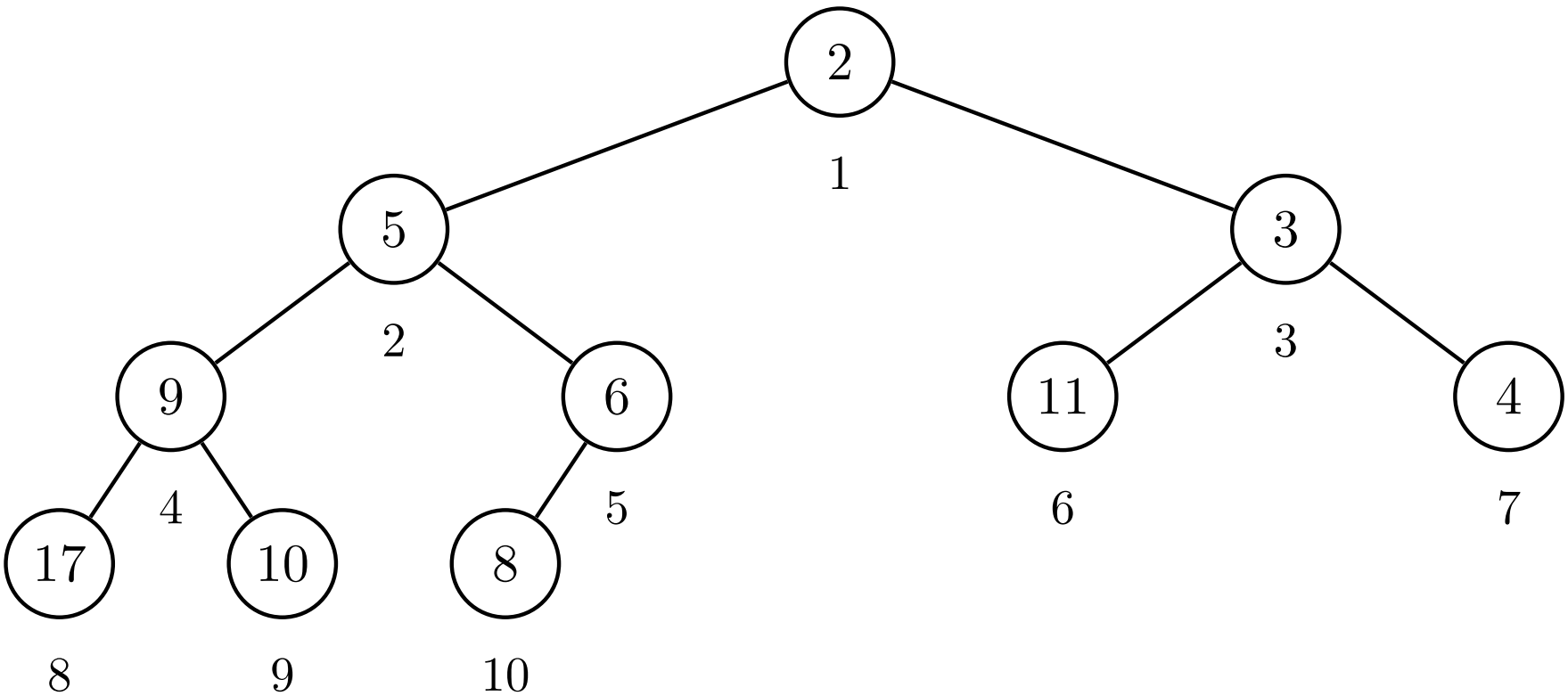
De ORDENINGSEIGENSCHAP VOOR BINAIRE HOPEN *zegt dat de sleutel van elke top hoogstens gelijk is aan de sleutel van zijn kinderen.*

Voorbeeld



Implementatie binaire hoop m.b.v. arrays

Omdat de binaire boom compleet is, is het *niet* nodig om effectief referenties naar kinderen e.d. te gaan bijhouden.
We 'nummeren' de toppen als niveau per niveau, en van links naar rechts. Dus bv.



Verband rangnummer ouder/kind

Eigenschap

Wanneer een top rangnummer i heeft, dan hebben zijn linker- en rechterkind (als die bestaan) respectievelijk rangnummer $2i$ en $2i + 1$.
Omgekeerd geldt: wanneer een top rangnummer i heeft (en deze top is niet de wortel van de boom), dan heeft zijn ouder rangummer $\text{floor}(i/2)$. We kunnen dus stellen dat:

$$\text{left}(i) = 2i,$$
$$\text{right}(i) = 2i + 1,$$
$$\text{parent}(i) = \text{floor}(i/2).$$

Opzoeken van element met kleinste sleutel

Uit de ordeningseigenschap volgt dat het kleinste element steeds in de wortel zit.

Het opzoeken van het kleinste element is dus een triviale bewerking.

Toevoegen van een element

1. Creëer een nieuw element.
2. Voeg dit element toe op de eerste beschikbare plaats. Dit betekent dus als een nieuw blad, met rangnummer i, waarbij we er steeds voor zorgen dat het diepste niveau gevuld is van links naar rechts.
3. Op dit moment is het in het algemeen zo dat de ordeningseigenschap voor binaire bomen nu kan geschonden zijn tussen i en $\mathrm{parent}(i)$. Indien dit zo is, verwissel dan i en $\mathrm{parent}(i)$. Dit herstelt de ordeningseigenschap tussen i en zijn ouder. Eventueel is nu de ordeningseigenschap tussen $\mathrm{parent}(i)$ en $\mathrm{parent}(\mathrm{parent}(i))$ geschonden. Indien dit zo is wissel dan beide elementen. Ga zo verder tot de binaire hoop is hersteld.

Voorbeeld

Voeg sleutel 1 toe aan de binaire hoop die we reeds hebben gezien.

Hoeveel verwisselingen kunnen maximaal nodig zijn bij het OMHOOG BUBBELEN?

Verwijderen van het element met de kleinste sleutel

Wanneer we het kleinste element verwijderen, dan hebben we een 'gat' in de wortel. We vullen dit op met het laatste element. Dan moet de ordeningseigenschap hersteld worden:

1. Verwissel de wortel met het meest rechtse blad met de grootste diepte.
2. Verwijder het meest rechtse blad; de binaire hoop heeft nu een element minder.
3. Indien de ordeningseigenschap geschonden is in de wortel, herstel deze dan door de wortel en zijn kleinste kind i van plaats te verwisselen. Indien de ordeningseigenschap nu geschonden is in i, herstel ze dan door i te verwisselen met de kleinste van zijn kinderen. Ga zo verder tot de binaire hoop hersteld is.

Voorbeeld

Verwijder de kleinste sleutel uit de binaire hoop die we reeds hebben gezien.

Hoeveel verwisselingen kunnen maximaal nodig zijn bij het OMLAAG BUBBELEN?

Tijdscomplexiteit

1. Opzoeken van het kleinste element gebeurt in constante tijd.
2. Toevoegen en verwijderen neemt tijd $\Theta(\lg n)$. (Waarom?)

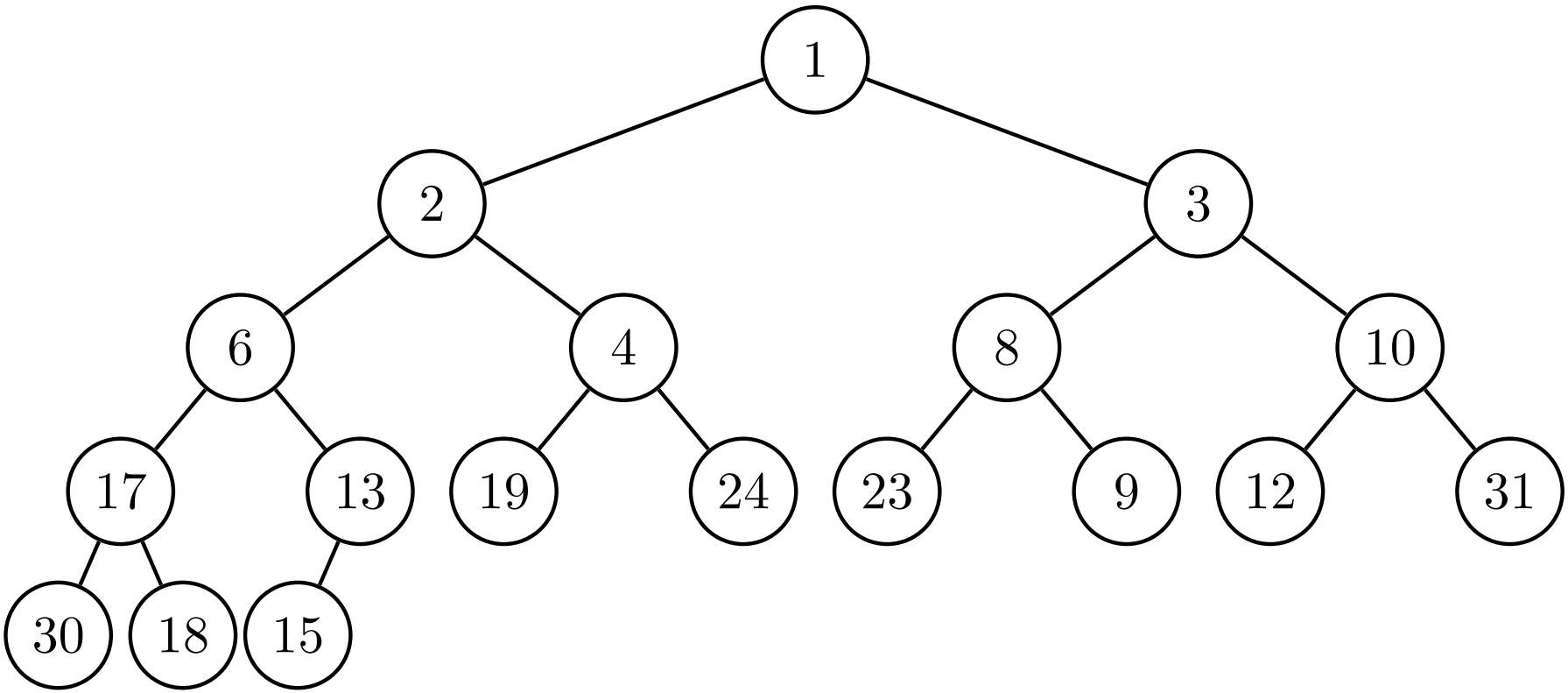
Oefeningen

1. Start met een lege binaire hoop. Voeg achtereenvolgens de volgende elementen toe aan de binaire hoop:

$$11, 13, 1, 15, 6, 5, 9, 16, 3, 10, 7, 4, 12, 14, 2.$$

   Teken de resulterende hoop na elke toevoeging.

Oefeningen



2. Beschouw de binaire hoop hierboven. Verwijder de drie kleinste elementen uit deze binaire hoop. Teken de binaire hoop na elke verwijdering.

Oefeningen

3. Wat worden de relaties in Eigenschap 4.36 wanneer een binaire hoop wordt opgeslaan in een array met als eerste index 0?
4. Waar kan het maximale element zich bevinden in een binaire hoop, aannemende dat de binaire hoop bestaat uit verschillende elementen.

Motivatie

In heel wat praktische situaties heeft men te maken met de situatie waarin 'objecten' verbonden zijn door een bepaalde relatie:

- steden zijn met elkaar verbonden m.b.v. wegen; kost: de afstand in kilometer,
- computers zijn verbonden m.b.v. netwerkkabels; kost: de communicatiesnelheid van de verbinding,
- luchthavens zijn met elkaar verbonden door directe vluchten; kost: de duur van de rechtstreekse vlucht.

Definitie

Definitie

Een GRAAF G *bestaat uit een verzameling* KNOPEN V*, en een verzameling* BOGEN E*. Elke boog verbindt twee knopen, en we noteren* $e = (v, w)$*. De graaf* G *wordt genoteerd als het koppel* (V, E)*, dus* $G = (V, E)$*.*

Opmerking:

- als $(v, w) \in E$, dan zijn v en w ADJACENT; v en w zijn INCIDENT met e (en omgekeerd)
- De BUREN van een knoop v zijn alle knopen w die adjacent zijn met v;
- aantal buren van een top: GRAAD van die knoop
- $\#V$, wordt de ORDE van de graaf genoemd; notatie n
- $\#E$, noemt men de GROOTTE; notatie m

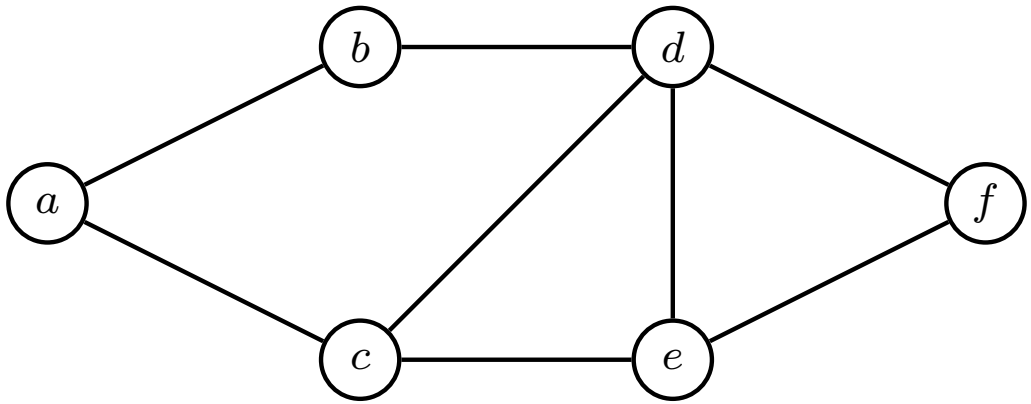
Gerichte versus ongerichte graaf

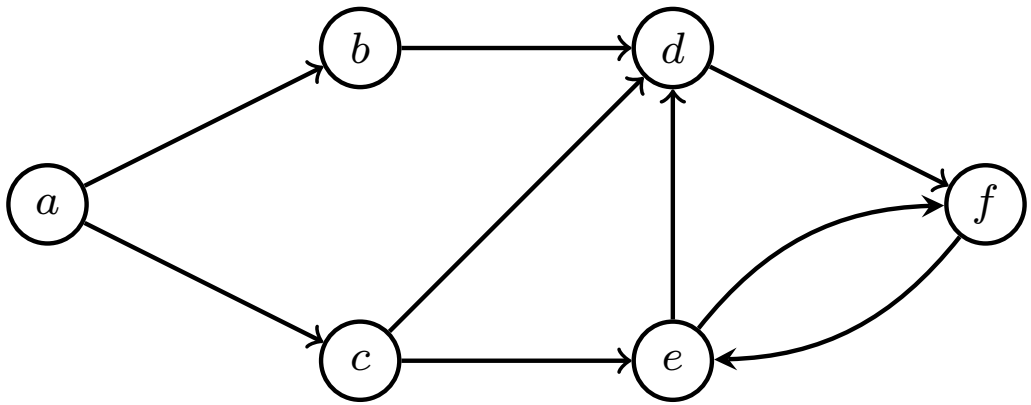
- Wanneer de boogparen niet geordend zijn dan spreekt men van een ONGERICHTE graaf.
- Wanneer de boogparen wel geordend zijn dan heeft men een GERICHTE graaf.
  In een gerichte graaf heeft een boog (v, w) een STAART v en een KOP w.
- GEWOGEN graaf: associeer getal met de bogen

Voorbeelden

- vriendschapsgraaf Facebook: ongericht
- volgersgraaf Twitter: gericht

Grafische voorstelling





Paden en cykels

Definitie

Een PAD *in een graaf G is een opsomming van toppen $(v_1, v_2, \ldots, v_k)$ zodanig dat er een boog bestaat tussen v_i en v_{i+1} voor $i \in \{1, 2, \ldots, k-1\}$. De* LENGTE *van dit pad is $k-1$, zijnde het aantal bogen op dit pad.*

Opmerking:

$k = 1$ is toegestaan: (v) is een pad van lengte nul.

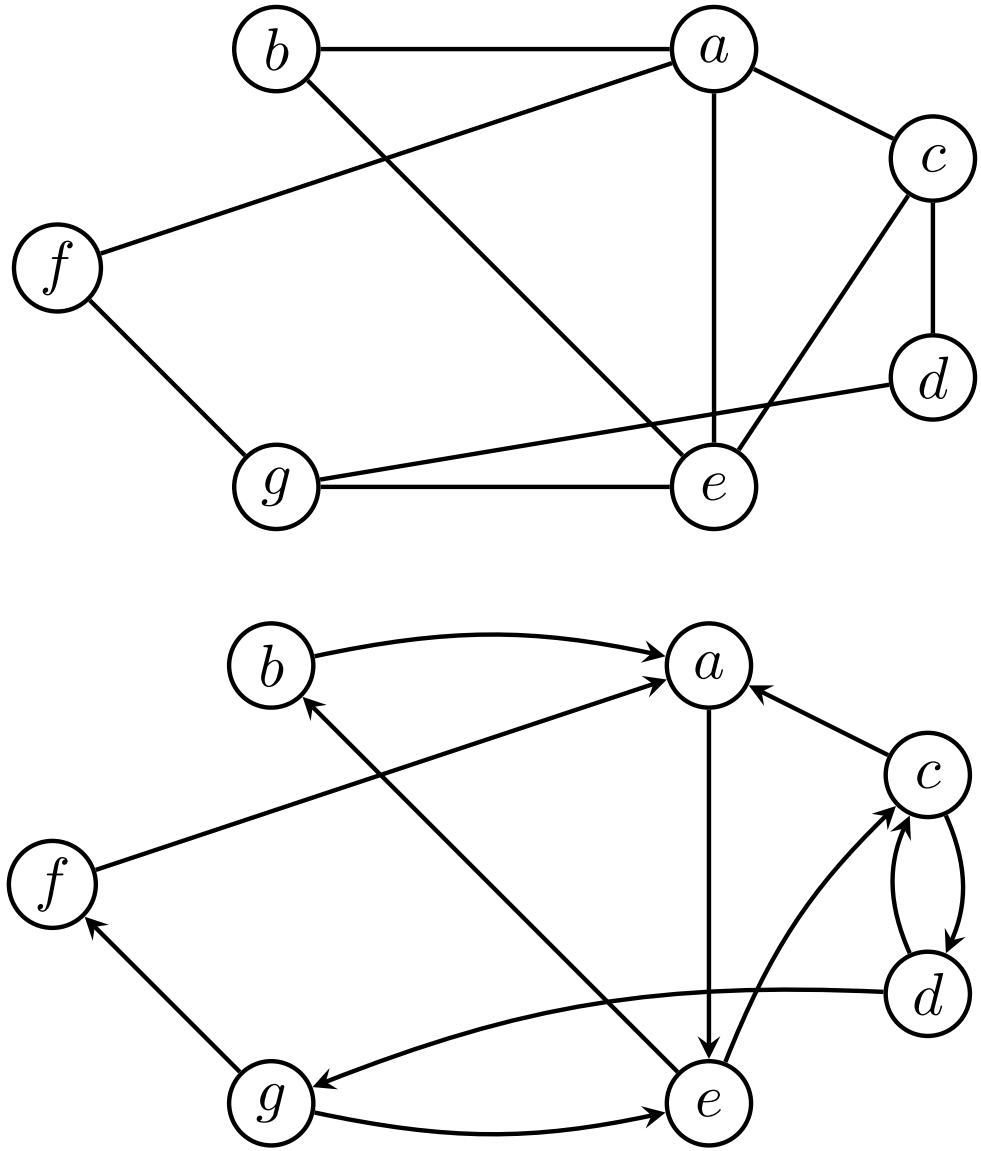
Definitie

Een ENKELVOUDIGE CYKEL *in een graaf is een pad waarvan de lengte strikt positief is en dat begint en eindigt in dezelfde knoop, waarbij alle knopen (behalve de start- en eindknoop) verschillend zijn en waarbij bovendien geen boog méér dan een keer wordt doorlopen.*

Voorbeeld

- In de ongerichte graaf is (a, b, d, c, e, f) een pad van lengte 5 van a naar f.
  Dit is *geen* pad in de gerichte graaf omdat er geen boog is van d naar c (enkel van c naar d).
- In beide grafen is het pad (d, f, e, d) een enkelvoudige cykel van lengte 3.
- In de gerichte graaf is (e, f, e) een enkelvoudige cykel van lengte 2, maar dit is *niet* zo in de ongerichte graaf omdat we de boog (e, f) twee keer zouden gebruiken in dit pad.
- In de ongerichte graaf is (a, b, d, f, e, d, c, a) een pad van lengte 7 van a naar a.
  Dit pad is *geen* enkelvoudige cykel want de knoop d wordt twee keer gebruikt!

Oefeningen



Oefeningen

1. Beschouw de gerichte en ongerichte graaf in op vorige slide (Figuur 5.3 in de cursus)
   - 1.1 Geef de bogenverzameling van deze twee grafen.
   - 1.2 Geef voor beide grafen de volgende verzameling $\text{buren}(e)$. Wat is de graad van de knoop e in beide gevallen?
   - 1.3 Vind het kortste pad (i.e. het pad met de kleinste lengte) van b naar d in beide grafen.
   - 1.4 Vind in beide grafen de langste enkelvoudige cykel die d bevat.

De adjacentiematrix

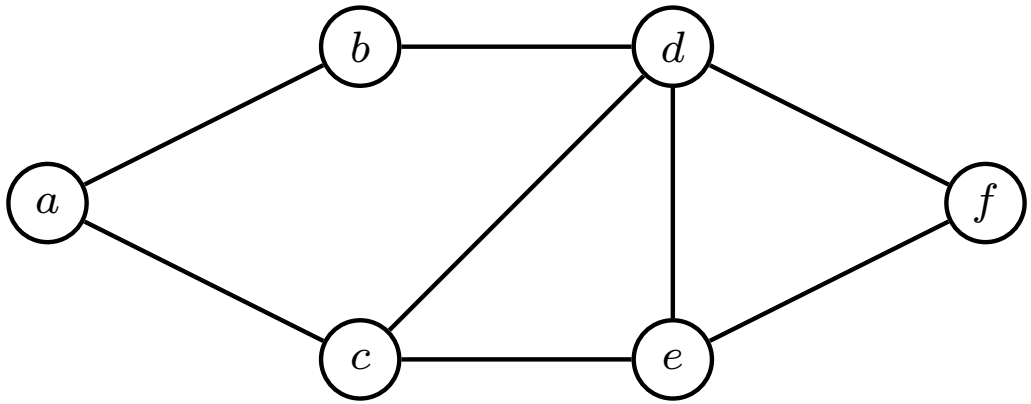
We veronderstellen dat de knopen van de graaf $G = (V, E)$ genummerd zijn van 1 t.e.m. n. Wanneer we te maken hebben met een (ongewogen) graaf dan kunnen we deze voorstellen door een ADJACENTIEMATRIX A. Voor deze adjacentiematrix geldt:

$$A_{i,j} = \begin{cases} 1 & \text{als } (i,j) \in E \\ 0 & \text{anders.} \end{cases}$$

Opmerking: Voor een ongerichte graaf is de adjacentiematrix steeds symmetrisch, i.e. A^T is steeds gelijk aan A.

Voorbeeld

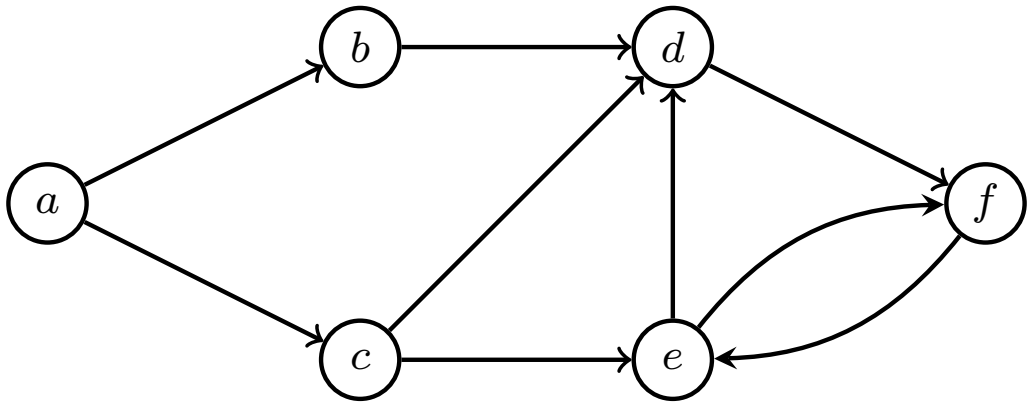
Beschouw de graaf:



Adjacentiematrix:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

Voorbeeld



Geef de adjacentiematrix van deze graaf. Wat merk je?

Adjacentiematrix kan ook gebruikt worden voor gewogen grafen.

Geheugen- en tijdsgebruik

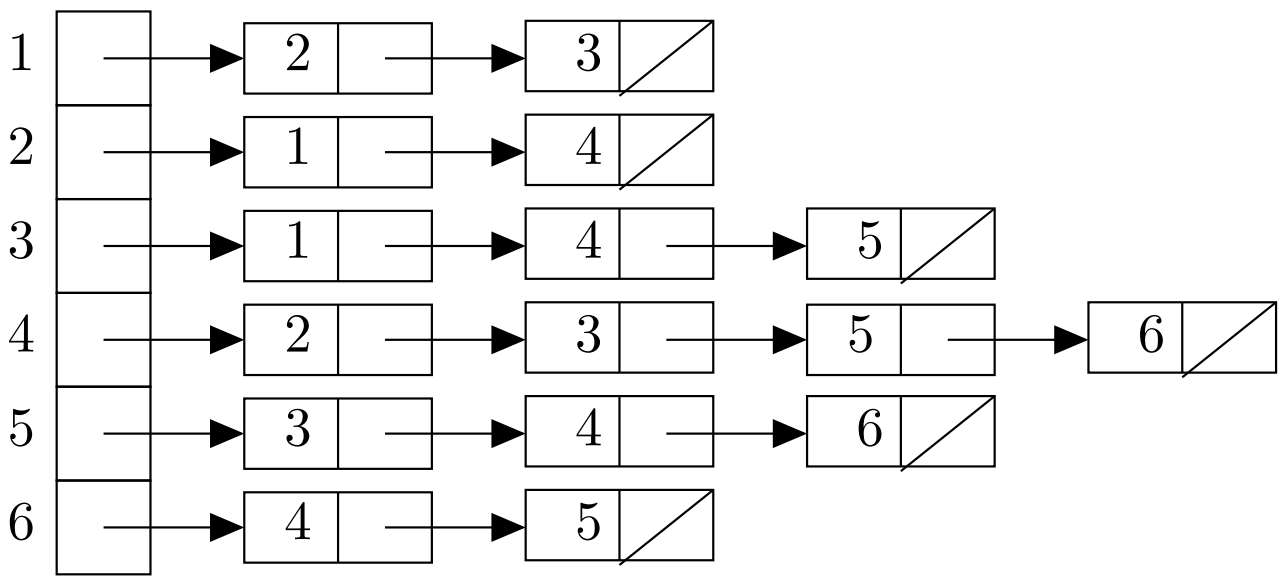
- Geheugenruimte steeds $\Theta(n^2)$.
- Maximaal aantal bogen in een graaf is $\Theta(n^2)$ (waarom?)
- In een IJLE graaf wordt zo heel wat geheugen verspild.
- Bepalen of twee knopen adjacent zijn: $\Theta(1)$.
- Alle buren bepalen/overlopen: $\Theta(n)$, onafhankelijk van de graad van de knoop.

Voorbeeld gelabelde en gewogen graaf

Brugge	$\to 1$	∞	57	∞	∞	∞	133	∞	∞	∞	∞	∞
Gent	$\to 2$	57	∞	55	58	86	114	∞	∞	∞	∞	∞
Brussel	$\to 3$	∞	55	∞	45	30	65	∞	∞	99	35	83
ntwerpen	$\to 4$	∞	58	45	∞	64	∞	∞	∞	∞	∞	79
Leuven	$\to 5$	∞	86	30	64	∞	94	∞	∞	76	25	58
Bergen	$\to 6$	133	114	65	∞	94	∞	77	∞	∞	68	∞
Namen	$\to 7$	∞	∞	∞	∞	∞	77	∞	130	63	38	∞
Aarlen	$\to 8$	∞	∞	∞	∞	∞	∞	130	∞	124	∞	∞
Luik	$\to 9$	∞	∞	99	∞	76	∞	63	124	∞	86	55
Waver	$\to 10$	∞	∞	35	∞	25	68	38	∞	86	∞	∞
Hasselt	$\to 11$	∞	∞	83	79	58	∞	∞	∞	55	∞	∞

Adjacentielijst-voorstelling

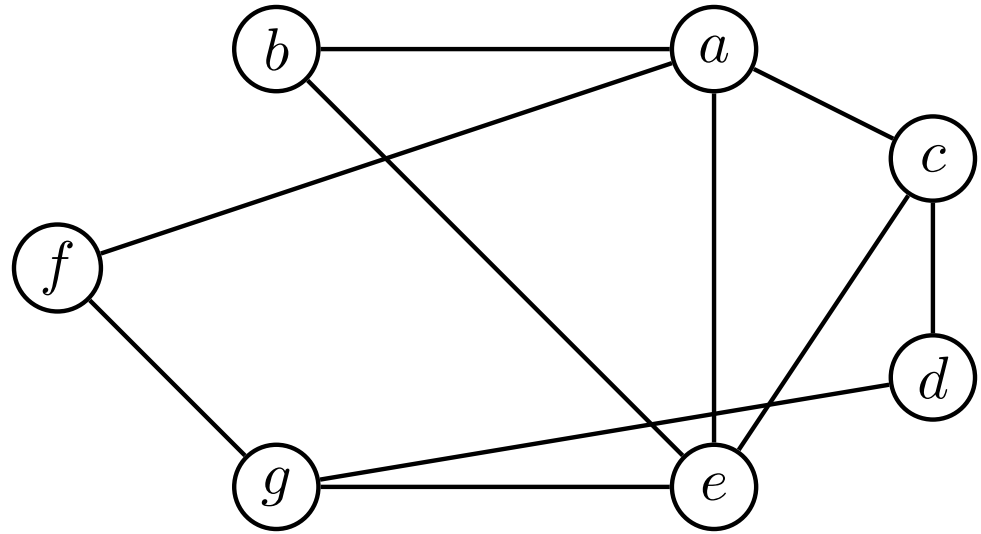
De ADJACENTIELIJST-VOORSTELLING van een graaf G bestaat uit een array van toppen, genummerd 1 t.e.m. n. Op de plaats i van deze array worden, in een lineair gelinkte lijst, de buren van top i bijgehouden.
Voor de voorbeeldgraaf:

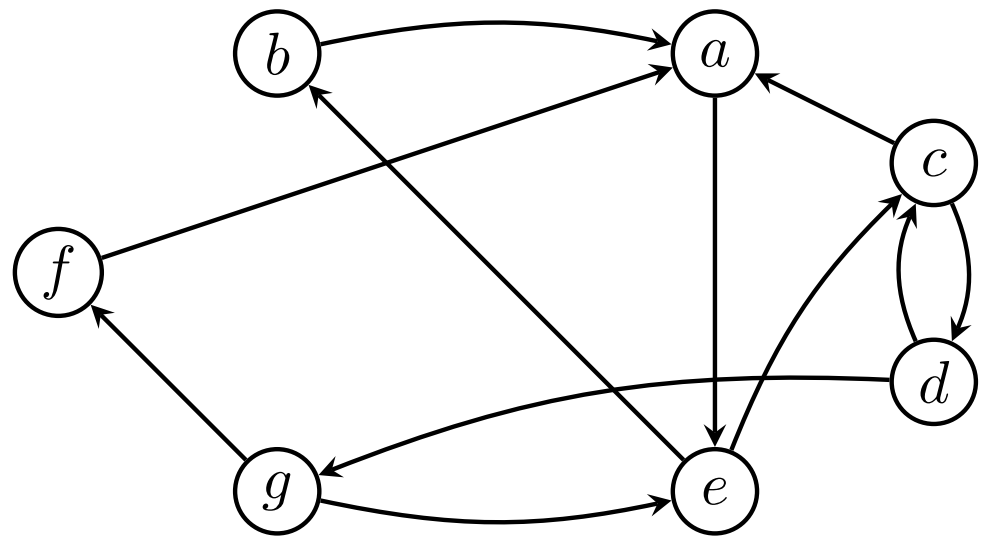


Geheugen- en tijdsgebruik

- De adjacentielijst-voorstelling gebruikt $\Theta(n + m)$ geheugenruimte.
- Het bepalen of twee knopen adjacent zijn kan nu *niet* langer in constante tijd gebeuren. Om te weten of i en j adjacent zijn moeten we immers de gelinkte lijst horend bij i overlopen om na te gaan of j in deze lijst aanwezig is.
- Als we alle buren van een knoop i willen overlopen dan gebeurt dit nu in een tijd die lineair is in het aantal buren van de knoop i. Dit is theoretisch de best mogelijke uitvoeringstijd.

Oefeningen





Oefeningen

1. Beschouw de gerichte en ongerichte graaf in Figuur 5.3 (vorige slide).
   - Geef voor beide grafen de adjacentiematrix. Je mag veronderstellen dat a rangnummer 1 heeft, b rangnummer 2 enzovoort.
   - Geef voor beide grafen de adjacentielijst-voorstelling. Je mag veronderstellen dat a rangnummer 1 heeft, b rangnummer 2 enzovoort.
2. Hoe berekent men de graad van een top i van een graaf G wanneer de adjacentiematrix A van de graaf gegeven is? Geef een algoritme. Wat is de tijdscomplexiteit van deze methode?
3. Hoe berekent men de graad van een top i van een graaf G wanneer de adjacentielijstvoorstelling van de graaf gegeven is? Geef een algoritme. Wat is de tijdscomplexiteit van deze methode?

Zoeken in Grafen: Inleiding

Veronderstel dat een graaf $G = (V, E)$ gegeven is. Dan kunnen we geïnteresseerd zijn om algoritmes te vinden die de volgende vragen kunnen beantwoorden:

1. Welke knopen kunnen we bereiken vanuit een knoop v?
2. Bestaat er een pad van v naar een specifieke knoop w?
3. Wat is het kortste pad van v naar w?

Generiek zoeken

- Start vanaf knoop s; vind alle knopen v waarvoor er een pad is van s naar v.
- Idee: initieel ontdekte gebied $=$ knoop s
- Breid in elke stap het ontdekte gebied uit door een boog (u, v) te volgen die de "grens" oversteekt.
- Op deze manier voeg je v toe aan het ontdekte gebied.
- Stop wanneer je geen bogen meer kan volgen, i.e. wanneer er geen bogen meer zijn die de grens oversteken.

Generiek zoeken: code

Invoer Een gerichte of ongerichte graaf $G = (V, E)$ met orde $n > 0$. Een knoop s waarvan het zoeken vertrekt. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een array D met $D[v] = \mathbf{true}$ als en slechts als er een pad bestaat van s naar v.

```
1: function ZoekGeneriek(G, s)
2:     D ← [false, false, ..., false]                    ▷ n keer false
3:     D[s] ← true                                        ▷ markeer s
4:     while ∃(u, v): D[u] = true ∧ D[v] = false do
5:         kies een boog (u, v) met D[u] = true ∧ D[v] = false
6:         D[v] ← true                                    ▷ markeer v
7:     end while
8:     return D
9: end function
```

Generiek Zoeken: eigenschap

Eigenschap

Wanneer het algoritme voor generiek zoeken eindigt dan geldt voor elke knoop v van G dat v gemarkeerd is als "ontdekt" (i.e. $D[v] = \mathbf{true}$) als en slechts als er een pad bestaat van s naar v in G.

Bewijs.

Twee delen: zie bord! □

Generiek zoeken: voorbeeld

Voorbeeldgraaf uit cursus.

gemarkeerde knopen	mogelijke bogen	gekozen boog
$\{1\}$	$(1,2),(1,3)$	$(1,2)$
$\{1,2\}$	$(1,3),(2,4)$	$(1,3)$
$\{1,2,3\}$	$(2,4),(3,4),(3,5)$	$(3,4)$
$\{1,2,3,4\}$	$(3,5),(4,5),(4,6)$	$(4,6)$
$\{1,2,3,4,6\}$	$(3,5),(4,5),(5,6)$	$(3,5)$
$\{1,2,3,4,5,6\}$	geen	geen

Breedte-Eerst Zoeken

Idee: bezoek de knopen in "lagen".

Eerst s zelf, dan de knopen die één boog verwijderd zijn van s, dan de knopen die twee bogen verwijderd zijn van s, enzovoort.

Gebruikte datastructuur: wachtrij (FIFO)

Breedte-Eerst Zoeken: pseudocode

Invoer Een gerichte of ongerichte graaf $G = (V, E)$ met orde $n > 0$.
Een startknoop s; $V = \{1, 2, \ldots, n\}$.

Uitvoer Een array D met $D[v] = \textbf{true}$ asa $\exists$ pad van s naar v.

```
 1: function BreedteEerst(G, s)
 2:     D ← [false, false, ..., false]                ▷ n keer false
 3:     D[s] ← true                                   ▷ markeer s
 4:     Q.init()                                      ▷ wachtrij van knopen
 5:     Q.enqueue(s)
 6:     while Q ≠ ∅ do
 7:         v ← Q.dequeue()
 8:         for all w ∈ buren(v) do
 9:             if D[w] = false then                  ▷ w nog niet ontdekt
10:                 D[w] ← true
11:                 Q.enqueue(w)
12:             end if
13:         end for
14:     end while
15:     return D
16: end function
```

Breedte-Eerst Zoeken: Voorbeeld

iteratie	Q	D
1	[1]	[T,F,F,F,F,F]
2	[2,3]	[T,T,T,F,F,F]
3	[3,4]	[T,T,T,T,F,F]
4	[4,5]	[T,T,T,T,T,F]
5	[5,6]	[T,T,T,T,T,T]
6	[6]	[T,T,T,T,T,T]
7	[]	[T,T,T,T,T,T]

Breedte-Eerst Zoeken: uitvoeringstijd

Eigenschap

Wanneer een adjacentielijst-voorstelling gebruikt wordt voor een graaf G, dan is de uitvoeringstijd $T(n, m)$ van Algoritme 5.2 van de grootte-orde $\Theta(n + m)$.

Oefening

1. Een ongerichte graaf is GECONNECTEERD als en slechts als er een pad bestaat tussen elke twee knopen v en w.
   - Ga na dat de bovenstaande definitie equivalent is met zeggen dat er een pad bestaat van een bepaalde knoop s naar alle andere knopen.
   - Schrijf een methode ISGECONNECTEERD die nagaat of een ongerichte graaf geconnecteerd is (return-waarde **true**) of niet (return-waarde **false**). Doe dit door de methode BREEDTEEERST aan te passen.

Diepte-Eerst Zoeken

Idee: zo snel mogelijk zo diep mogelijk in de graaf. (zie: http://xkcd.com/761/)

We bezoeken steeds de meest recent ontdekte knoop.

We gebruiken m.a.w. een LIFO structuur (stapel) i.p.v. een wachtrij.

We gebruiken de impliciete call-stack.

Diepte-Eerst Zoeken: code

Invoer Een gerichte of ongerichte graaf $G = (V, E)$ met orde $n > 0$. Een startknoop s; $V = \{1, 2, \ldots, n\}$.

Uitvoer Een array D met $D[v] = \textbf{true}$ asa $\exists$ pad van s naar v.

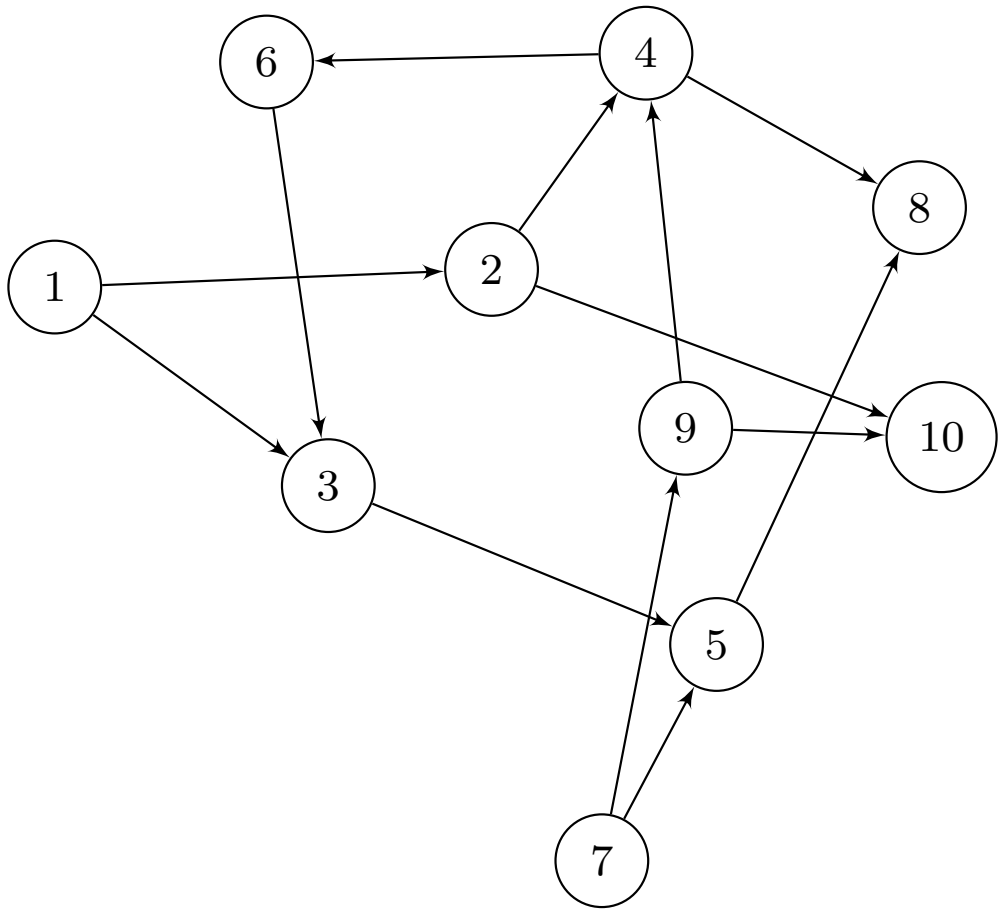
```
1:  function DiepteEerst(G, s)
2:      D ← [false, false, ..., false]                ▷ n keer false
3:      DiepteEerstRecursief(G, s, D)
4:      return D
5:  end function
6:  function DiepteEerstRecursief(G, v, D)
7:      D[v] ← true                                    ▷ markeer v
8:      for all w ∈ buren(v) do
9:          if D[w] = false then                       ▷ w nog niet ontdekt
10:             DiepteEersteRecursief(G, w, D)
11:         end if
12:     end for
13: end function
```

Diepte-Eerst Zoeken: Voorbeeld

DiepteEerstRecursief(G, 1, [**false**, **false**, **false**, **false**, **false**, **false**])
  DiepteEerstRecursief(G, 2, [**true**, **false**, **false**, **false**, **false**, **false**])
    DiepteEerstRecursief(G, 4, [**true**, **true**, **false**, **false**, **false**, **false**])
      DiepteEerstRecursief(G, 3, [**true**, **true**, **false**, **true**, **false**, **false**])
        DiepteEerstRecursief(G, 5, [**true**, **true**, **true**, **true**, **false**, **false**])
          DiepteEerstRecursief(G, 6, [**true**, **true**, **true**, **true**, **true**, **false**])

Topologisch sorteren

Precedentiegraaf van software modules.



In welke volgorde kunnen de modules gecompileerd worden?

Topologische Sortering: definitie

Definitie

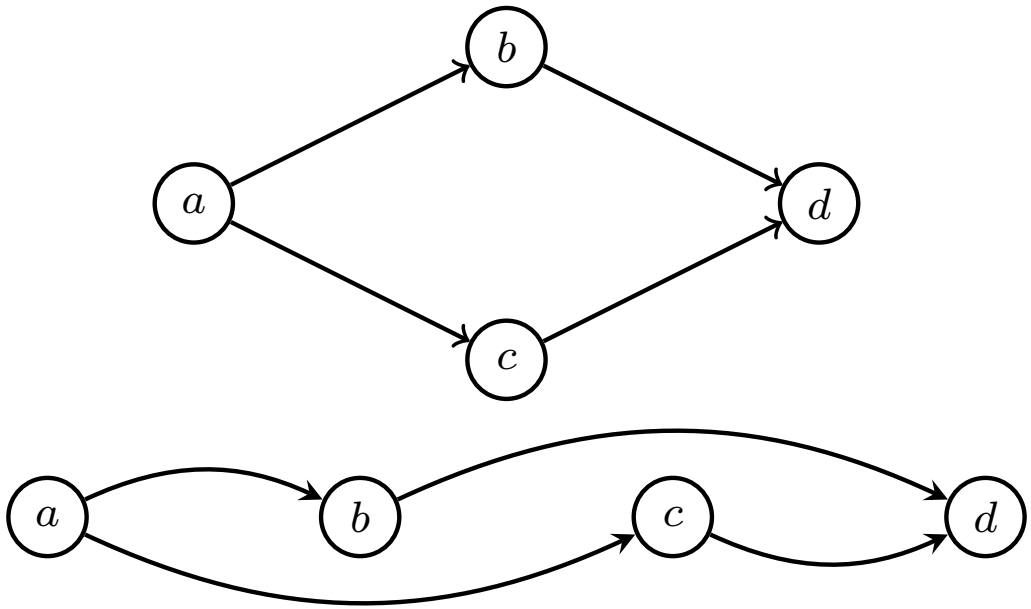
Een TOPOLOGISCHE SORTERING *van een gerichte graaf G kent aan elke knoop v een verschillend rangnummer $f(v)$ toe van 1 t.e.m. n zodanig dat de volgende eigenschap geldt:*

$$\forall (u, v) \in E \colon f(u) < f(v), \tag{2}$$

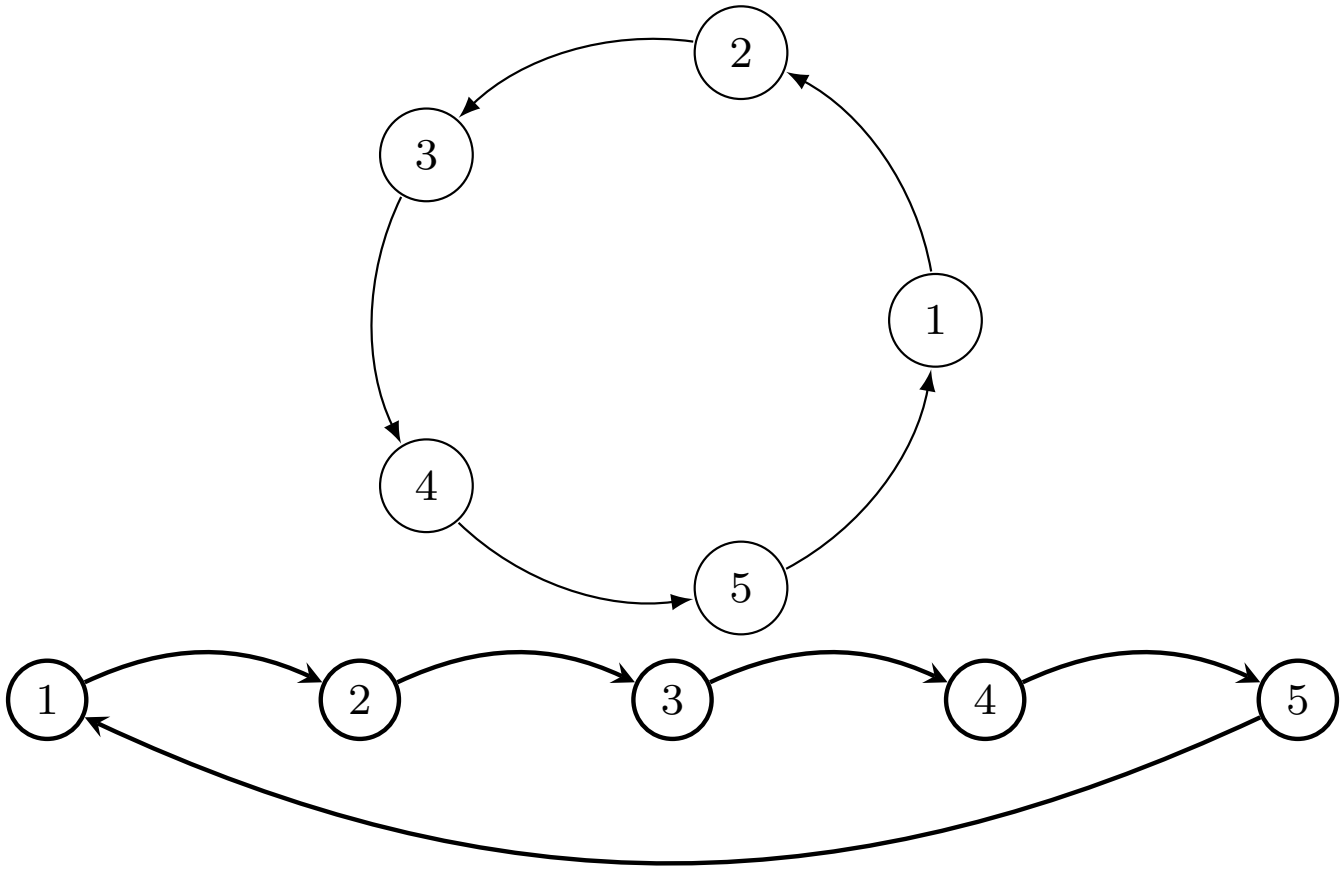
m.a.w. als (u, v) een boog is in de graaf dan is het rangnummer van de kop v groter dan het rangnummer van de staart u.

Wanneer we alle knopen op een rechte lijn tekenen, gesorteerd volgens het rangnummer van hun topologische sortering, dan zullen alle bogen vooruit wijzen.

Topologische Sortering: Voorbeeld



Topologische Sortering: (Mislukt) Voorbeeld



Topologische Sortering: Eigenschap

Eigenschap

Wanneer een gerichte graaf G geen enkelvoudige cykels heeft, dan bestaat er een topologische sortering van G.

Bewijs.

- Een gerichte graaf G zonder cykels heeft een knoop zonder buren.
- De knoop zonder buren is een goede kandidaat om rangnummer n te krijgen.
- Eenvoudig algoritme om topologische sortering te vinden.

□

Topologisch Sorteren: Algoritme

We kunnen *diepte-eerst zoeken* aanpassen om een topologische sortering te geven.

Idee: DFS zoekt vanuit elke knoop v alle knopen w waarvoor er een pad van v naar w bestaat. In een topologische sortering moet v dus vóór w komen.

Hou een lijst S bij; wanneer v is 'afgewerkt' met DFS, voeg dan v vooraan toe.

DFS moet eventueel meerdere malen worden opgeroepen!

Topologisch Sorteren: Ontdekken cykel

- Initieel alle knopen op 0 (*onontdekt*)
- Volledig afgewerkt knopen op 2 (*afgewerkt*)
- Knopen waarvoor we nog zoeken naar opvolgers op 1 (*bezig*)

Stel we zijn bezig met de buren van knoop v (dus v op 1). We vinden buur w met toestand 'bezig'. Dit betekent dat er een pad is van w naar v; samen met de boog (v, w) wordt dit een cykel!

Topologisch Sorteren: code

Invoer Een gerichte graaf $G = (V, E)$ met orde $n > 0$. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een topologische sortering van G indien mogelijk, **false** anders.

```
1:  function SorteerTopologisch(G)
2:      global cycleDetected ← false                    ▷ globale variabele
3:      D ← [0, 0, ..., 0]                              ▷ n keer 0
4:      S ← ∅                                           ▷ S is lege lijst
5:      for all s ∈ V do
6:          if D[s] = 0 then                            ▷ s nog niet gezien
7:              DfsTopo(G, s, D, S)   ▷ S en D referentieparameters
8:              if cycleDetected = true then            ▷ controleer op cykel
9:                  return false
10:             end if
11:         end if
12:     end for
13:     return S
14: end function
```

Code: vervolg

```
 1: function DfsTopo(G, v, D, S)
 2:     D[v] ← 1                                          ▷ markeer v als 'bezig'
 3:     for all w ∈ buren(v) do
 4:         if D[w] = 0 ∧ cycleDetected = false then      ▷ w nog niet ontdekt
 5:             DfsTopo(G, w, D, S)
 6:         else if D[w] = 1 then                         ▷ cykel ontdekt w ⇝ v → w
 7:             cycleDetected ← true
 8:         end if
 9:     end for
10:     D[v] ← 2                                          ▷ markeer v als 'voltooid'
11:     voeg v vooraan toe aan S                          ▷ ken rangnummer toe aan v
12: end function
```

Voorbeeld: softwaremodules

Uitwerking op het bord.

Oefeningen

2. Vind alle mogelijke topologische sorteringen van de graaf in Figuur 5.10.
3. Vind de compilatievolgorde van de modules in Figuur 5.9 wanneer de labels in dalende volgorde worden doorlopen.
4. Veronderstel nu dat er in de graaf van Figuur 5.9 een extra boog $(8, 6)$ wordt toegevoegd. Pas nu het algoritme voor topologisch sorteren toe.

Kortste Pad in een Ongewogen Graaf

We wensen te weten hoeveel stappen (bogen) men nodig heeft om, startend vanaf een top s, een andere top v te bereiken.

We zoeken m.a.w. de lengte van een kortste pad.

Breedte-Eerst Zoeken overloopt de graaf 'laag per laag'. Kleine aanpassing nodig om kortste pad bij te houden.

Kortste Pad in een Ongewogen Graaf: pseudocode

Invoer Een gerichte of ongerichte ongewogen graaf $G = (V, E)$. Een startknoop s; $V = \{1, 2, \ldots, n\}$.

Uitvoer De array D met $D[v]$ de kortste afstand van s tot v; als $D[v] = \infty$ dan is er geen pad van s naar v.

```
 1: function KortstePadOngewogen(G, s)
 2:     D ← [∞, ∞, ..., ∞]                                  ▷ n keer ∞
 3:     D[s] ← 0              ▷ kortste pad van s naar zichzelf heeft lengte 0
 4:     Q.init()                                            ▷ wachtrij van knopen
 5:     Q.enqueue(s)
 6:     while Q ≠ ∅ do
 7:         v ← Q.dequeue()
 8:         for all w ∈ buren(v) do
 9:             if D[w] = ∞ then                            ▷ w nog niet ontdekt
10:                 D[w] ← D[v] + 1
11:                 Q.enqueue(w)
12:             end if
13:         end for
14:     end while
15:     return D
16: end function
```

Kortste Pad in een Ongewogen Graaf: Voorbeeld

	1	2	3	4	5	6
(a)	0	∞	∞	∞	∞	∞
(b)	0	1	1	∞	∞	∞
(c)	0	1	1	2	∞	∞
(d)	0	1	1	2	2	∞
(e)	0	1	1	2	2	3

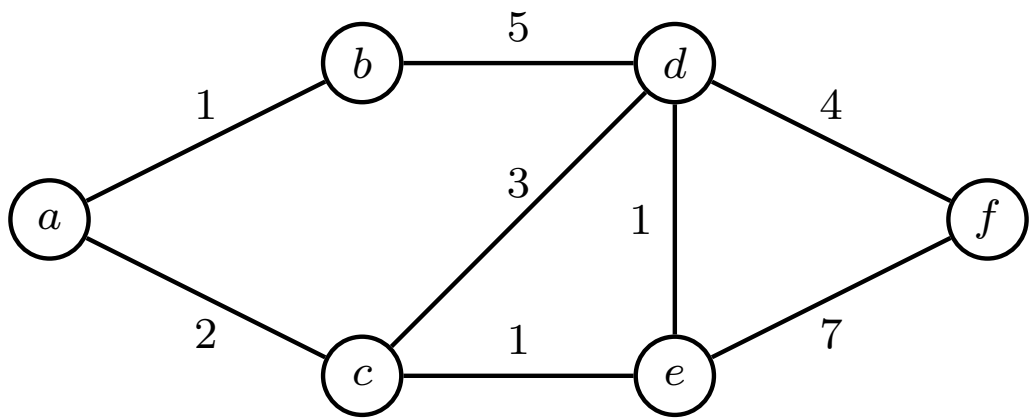
Kortste Pad in een Gewogen Graaf

In een *gewogen* graaf willen we meestal pad met kleinste gewicht (en niet noodzakelijk met het minst aantal bogen).
Naïeve aanpassing van breedte-eerst:

$$D[w] \leftarrow D[v] + \text{gewicht}(v, w).$$

(i.p.v. $D[w] \leftarrow D[v] + 1$)

Toepassing naïeve aanpassing



Wat wordt de afstandenarray?

Kortste Pad in een Gewogen Graaf: Dijkstra

Sleutelideeën:

1. Op elk moment: een verzameling S van knopen v waarvoor de kortste afstand van s tot v reeds gekend is, en een verzameling Q van knopen waarvoor de kortste afstand nog *niet* met zekerheid gekend is.
2. Voor elke knoop v (die tot Q behoort): $D[v]$ is de kortste afstand van een pad van s naar v *dat enkel uit knopen van S bestaat* (behalve de laatste).
3. We voegen telkens dié knoop v van Q toe aan S waarvoor $D[v]$ minimaal is onder alle knopen van Q. Dit betekent dat voor de buren w van v die tot Q behoren we eventueel $D[w]$ moeten aanpassen. Het pad van s naar v (dat nu enkel uit knopen van S bestaat) uitgebreid met de boog (v, w) zou eventueel korter kunnen zijn dan het tot dan toe gevonden kortste pad.

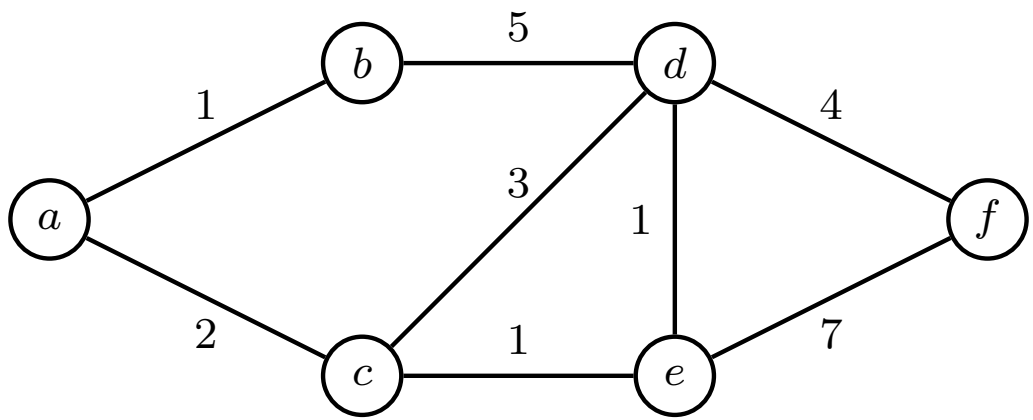
Dijkstra:Code

Invoer Een gewogen graaf $G = (V, E)$ met positieve gewichten.
Startknoop s; $V = \{1, 2, \ldots, n\}$.

Uitvoer De array D met $D[v]$ de kortste afstand van s tot v; als $D[v] = \infty$ dan is er geen pad van s naar v.

```
 1: function DIJKSTRA(G, s)
 2:     D ← [∞, ∞, ..., ∞]                                   ▷ n keer ∞
 3:     D[s] ← 0          ▷ kortste pad van s naar zichzelf heeft lengte 0
 4:     Q ← V      ▷ knopen waarvan kortste afstand nog niet is bepaald
 5:     while Q ≠ ∅ do
 6:         zoek v ∈ Q waarvoor D[v] minimaal is (voor knopen in Q)
 7:         verwijder v uit Q
 8:         for all w ∈ buren(v) ∩ Q do
 9:             if D[w] > D[v] + gewicht(v, w) then
10:                 D[w] ← D[v] + gewicht(v, w)      ▷ korter pad s → w
11:             end if
12:         end for
13:     end while
14:     return D
15: end function
```

Dijkstra: Voorbeeld



Pas Dijkstra toe met startknoop a.

Implementatie

In Dijkstra moeten herhaaldelijk minima worden berekend: prioriteitswachtrij (bv. binaire hoop)!

Probleem: sleutels moeten worden aangepast (verminderd).

Standaard niet mogelijk met een binaire hoop. Uitbreiding mogelijk m.b.v. tweede array.

Oefeningen

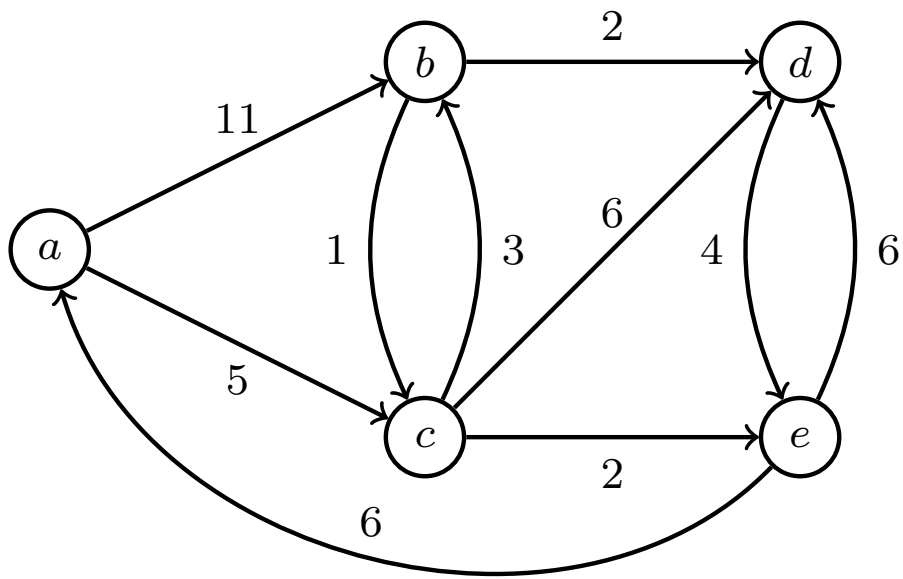
1. In Algoritme 5.5 wordt nu enkel de afstand van elke knoop v tot de startknoop s bijgehouden. In veel toepassingen heeft men echter ook een pad nodig dat deze minimale afstand realiseert.
   - Pas de pseudo-code van Algoritme 5.5 aan zodanig dat er een tweede array P wordt teruggeven zodanig dat $P[v]$ de knoop geeft die de voorganger is van v op een kortste pad van s naar v.
   - Pas je aangepaste algoritme toe op de ongerichte graaf in Figuur 5.9 startend vanaf knoop 1. Ga ervan uit dat knopen steeds worden bezocht in stijgende volgorde. Hoe zit de array P er uit na afloop?
   - Schrijf een algoritme dat als invoer de array P neemt en een knoop v. Het algoritme geeft een lijst terug die het kortste pad van s naar v bevat (in de juiste volgorde).

Oefeningen

2. Beschrijf hoe je volgend probleem kan oplossen als een kortste pad probleem. Gegeven een lijst van Engelstalige 5-letterwoorden. Woorden worden *getransformeerd* door juist één letter van het woord te vervangen door een andere letter. Geef een algoritme dat nagaat of een woord w_1 omgezet kan worden in een woord w_2. Indien dit het geval is dan moet je algoritme ook de tussenliggende woorden tonen voor de kortste sequentie van transformaties die w_1 in w_2 omzet.
3. Vind voor de graaf in Figuur 5.4 de lengte van het kortste pad van Brugge naar alle andere steden. Voer hiertoe het algoritme van Dijkstra uit.

Oefeningen

4. Vind voor de graaf in Figuur 5.14 (de lengte van) het kortste pad van de knoop a naar alle andere knopen. Voer hiertoe het algoritme van Dijkstra uit (en houd ook bij wat de kortste paden zijn).

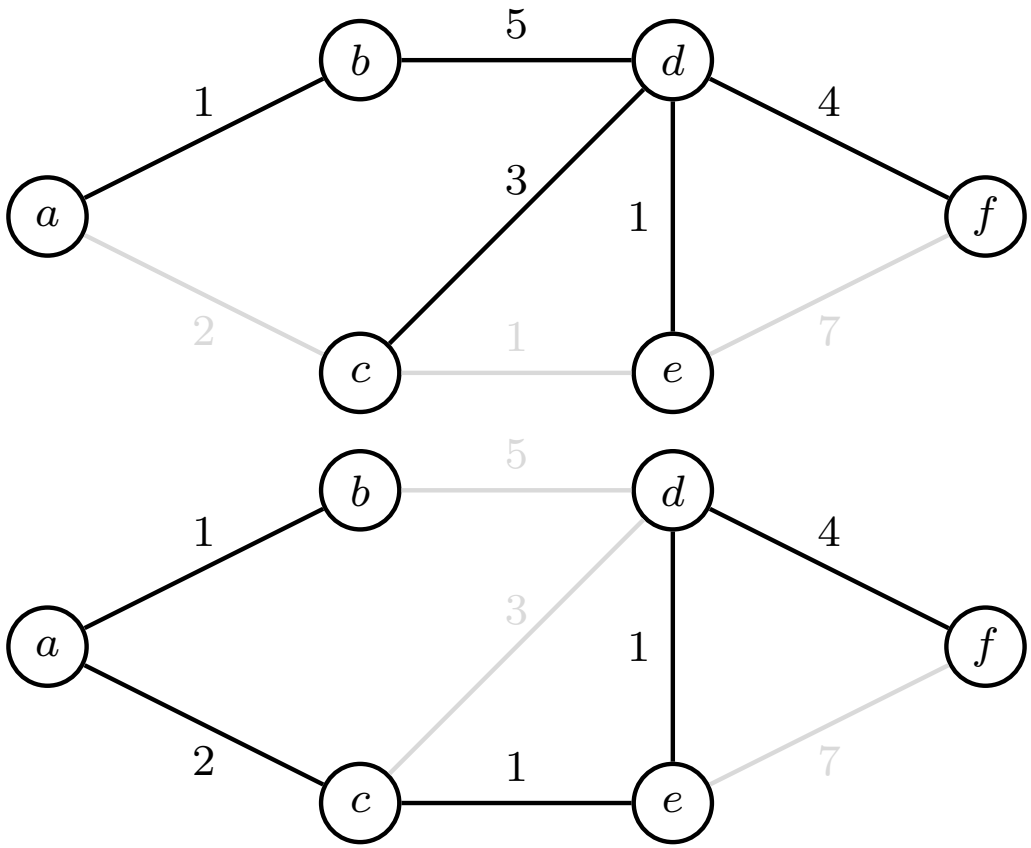


Opspannende Boom: Definitie

Definitie

Een OPSPANNENDE BOOM *van een ongerichte graaf* $G = (V, E)$ *is een verzameling van bogen* T*, met* $T \subseteq E$*, zodanig dat* $G' = (V, T)$ *een pad heeft tussen elke twee knopen van* V*, en zodanig dat* G' *geen enkelvoudige cykels heeft.*

Opspannende bomen: voorbeeld



Minimale Kost Opspannende Boom

Het gewicht van een boom is de som van de gewichten van zijn bogen:

$$\text{gewicht}(T) = \sum_{t \in T} \text{gewicht}(t).$$

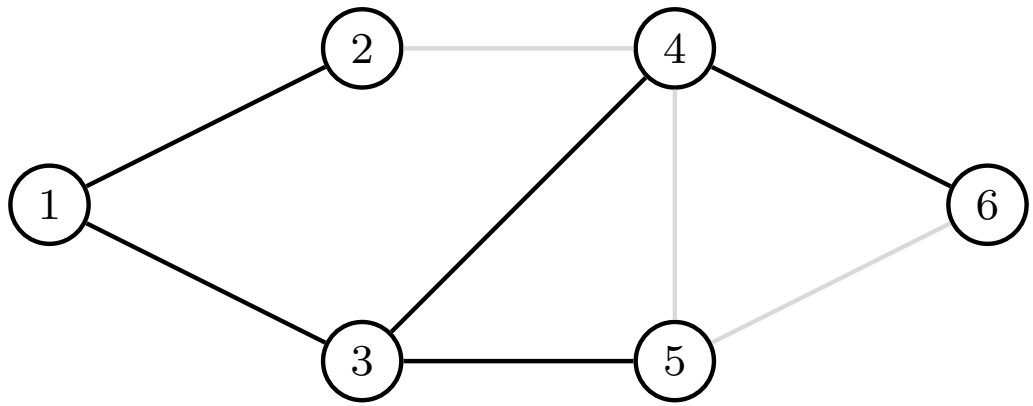
Definitie

Een MINIMALE KOST OPSPANNENDE BOOM T *van de graaf* G *is een opspannende boom zodanig dat voor alle andere opspannende bomen* T' *van* G *geldt dat*

$$\sum_{t \in T} \text{gewicht}(t) \leq \sum_{t' \in T'} \text{gewicht}(t').$$

Algoritme van Prim

Het algoritme voor generiek zoeken levert reeds een opspannende boom! (als we de bogen zouden bijhouden).
Dit is bijvoorbeeld een mogelijke uitvoer van generiek zoeken:



Essentie Prim: doe generiek zoeken maar kies steeds de *goedkoopste* beschikbare boog. Een *gulzige* strategie.

Algoritme van Prim: Code

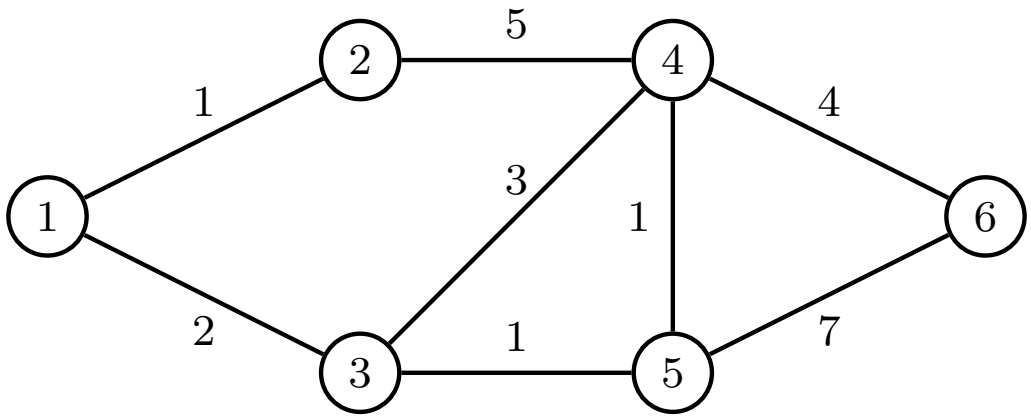
Invoer Een ongerichte gewogen graaf $G = (V, E)$ met orde $n > 0$. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een verzameling T van bogen die een minimale kost opspannende boom is.

```
 1: function PRIM(G)
 2:     D ← [false, false, ..., false]                    ▷ n keer false
 3:     D[1] ← true                          ▷ kies knoop 1 als startknoop
 4:     T ← ∅                                              ▷ gekozen bogen
 5:     while ∃(u, v): D[u] = true ∧ D[v] = false do
 6:         kies (u, v) met D[u] = true ∧ D[v] = false met minimaal gewicht
 7:         D[v] ← true
 8:         T ← T ∪ {(u, v)}                ▷ Voeg boog (u, v) toe aan boom
 9:     end while
10:     return T
11: end function
```

Algoritme van Prim: Voorbeeld

Voer het algoritme van Prim uit voor de volgende voorbeeldgraaf.



Algoritme van Kruskal

Alternatief algoritme voor vinden minimale kost opspannende boom.

Eveneens een gulzig algoritme. We kiezen steeds de goedkoopste boog.

Bogen zijn niet steeds met elkaar verbonden.

Boog wordt enkel gekozen indien die *geen cykel* veroorzaakt.

Algoritme van Kruskal: Code

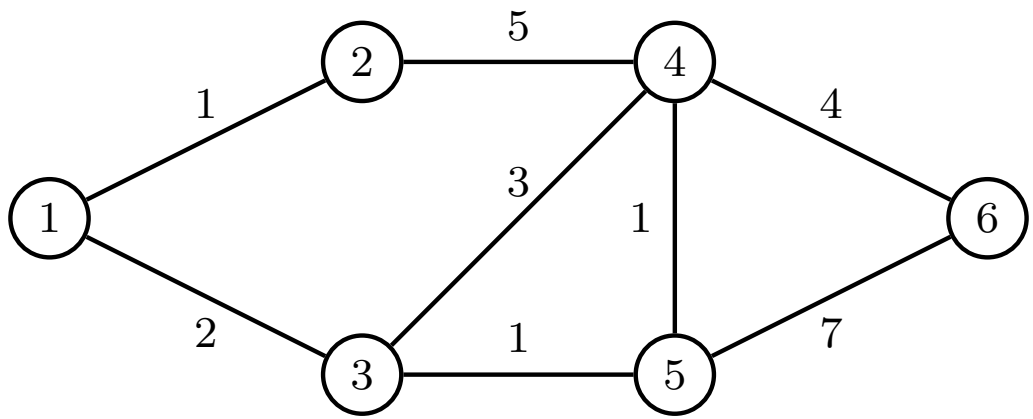
Invoer Een ongerichte gewogen graaf $G = (V, E)$ met grootte $n > 0$. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een verzameling T van bogen die een minimale kost opspannende boom is.

```
1:  function KRUSKAL(G)
2:      T ← ∅                                   ▷ Start met lege boom
3:      E' ← sorteer E volgens stijgend gewicht
4:      for all e' ∈ E' do
5:          if T ∪ e' heeft geen cykel then
6:              T ← T ∪ e'
7:          end if
8:      end for
9:      return T
10: end function
```

Algoritme van Kruskal: Voorbeeld

Voer het algoritme van Kruskal uit voor de volgende voorbeeldgraaf.



Oefeningen

1. Vind een minimale kost opspannende boom m.b.v. het algoritme van Prim voor de graaf in Figuur 5.4. Neem als startknoop “Brugge”.
2. Vind een minimale kost opspannende boom m.b.v. het algoritme van Kruskal voor de graaf in Figuur 5.4.

Probleemdefinitie

Definitie

Het HANDELSREIZIGERSPROBLEEM *is het volgende: gegeven een complete*[1] *gewogen ongerichte graaf G met niet-negatieve gewichten, vind dan een ordening van de knopen zodanig dat elke knoop juist éénmaal wordt bezocht (behalve de start- en eindknoop die samenvallen) en zodanig dat de som van de gewichten van de gekozen bogen minimaal is.*

[1] Een graaf is compleet als er een boog is tussen elke twee (verschillende) knopen.

Moeilijkheidsgraad

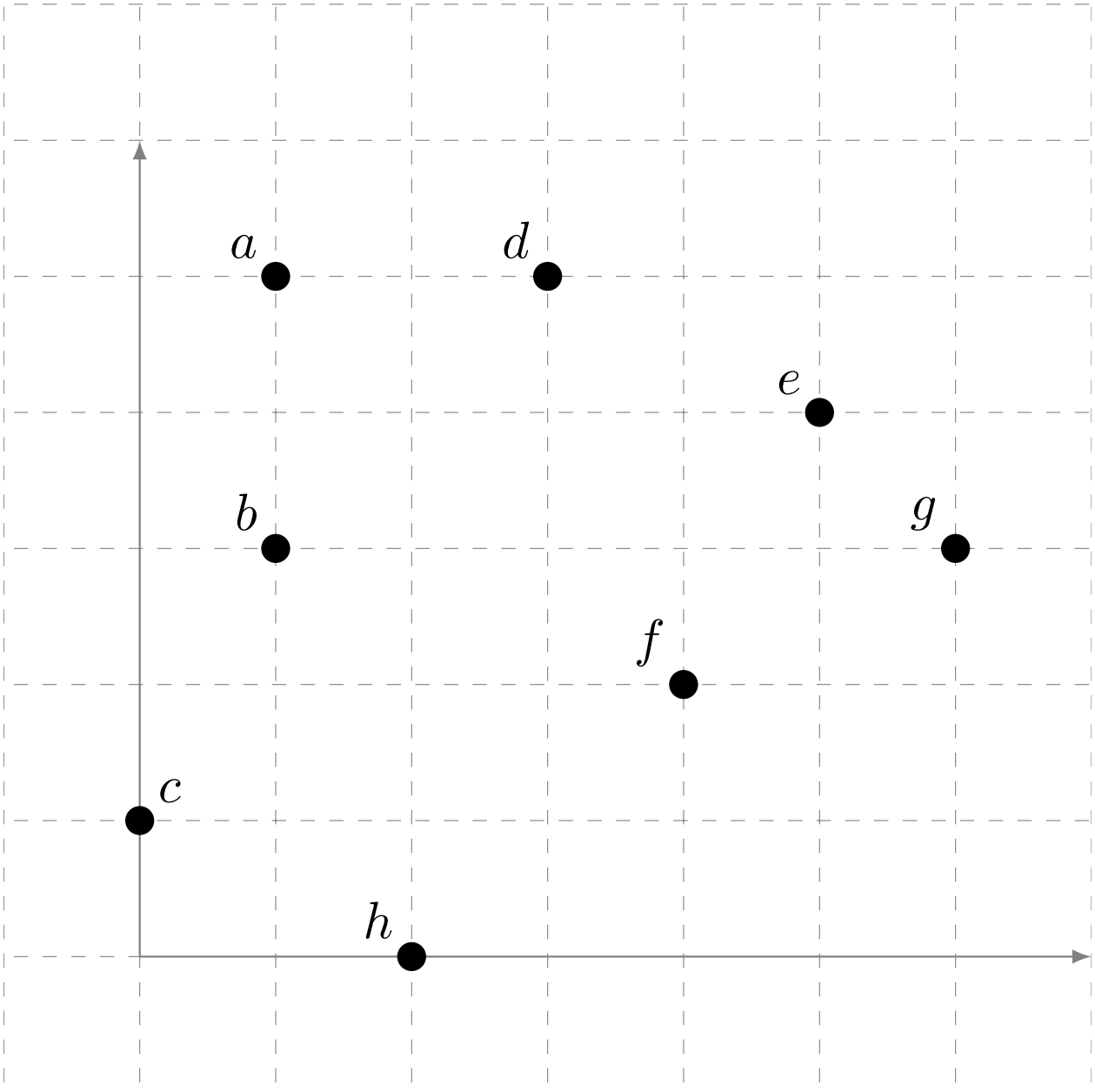
Voor alle vorige problemen hebben we steeds een efficiënt (polynomiaal) algoritme kunnen geven.

Het handelsreizigersprobleem is echter een NP-COMPLEET probleem, wat hoogstwaarschijnlijk betekent dat er *geen* polynomiaal algoritme bestaat dat alle gevallen correct kan oplossen.

Triviaal algoritme: probeer alle mogelijkheden en selecteer de beste. Aantal mogelijkheden is echter $(n-1)!$ (Je kan de eerste stad steeds als 'vast' beschouwen.)

Men gebruikt dan ook vaak *benaderende* algoritmen.

Steden op een grid



Driehoeksongelijkheid

Voor de vorige graaf is het steeds korter om rechtstreeks van v naar w te gaan dan om een omweg te maken via u. De graaf voldoet aan de driehoeksongelijkheid.

Definitie

Een graaf G voldoet aan de DRIEHOEKSONGELIJKHEID *wanneer voor alle knopen u, v en w geldt dat*

$$\text{gewicht}(v, w) \leq \text{gewicht}(v, u) + \text{gewicht}(u, w).$$

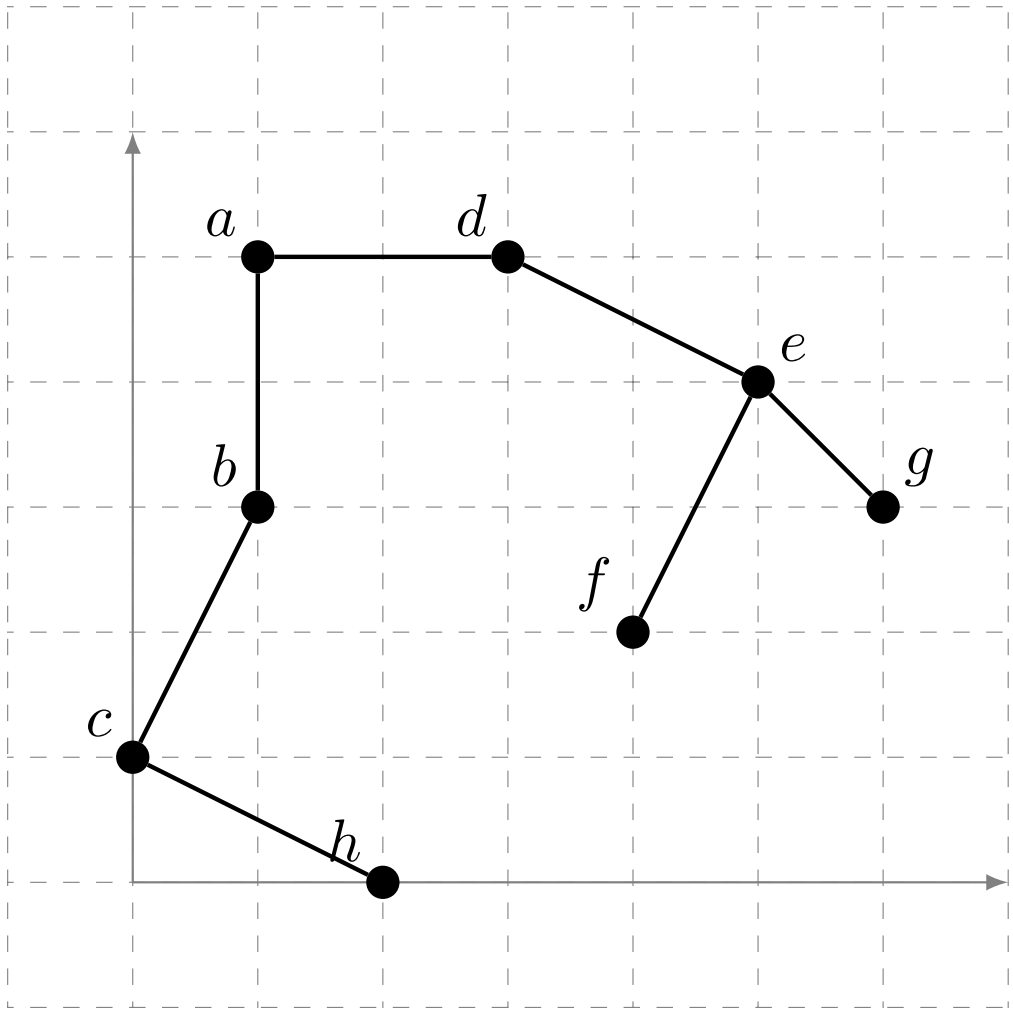
Benaderingsalgoritme voor handelsreizigersprobleem

1. Bereken een minimale kost opspannende boom T voor de graaf.
2. Kies willekeurig een wortel r van deze boom.
3. Geef de cykel terug die correspondeert met het in *preorde* doorlopen van deze boom.

Wanneer de graaf G aan de driehoeksongelijkheid voldoet dan is de gevonden oplossing hoogstens tweemaal zolang als de optimale oplossing.

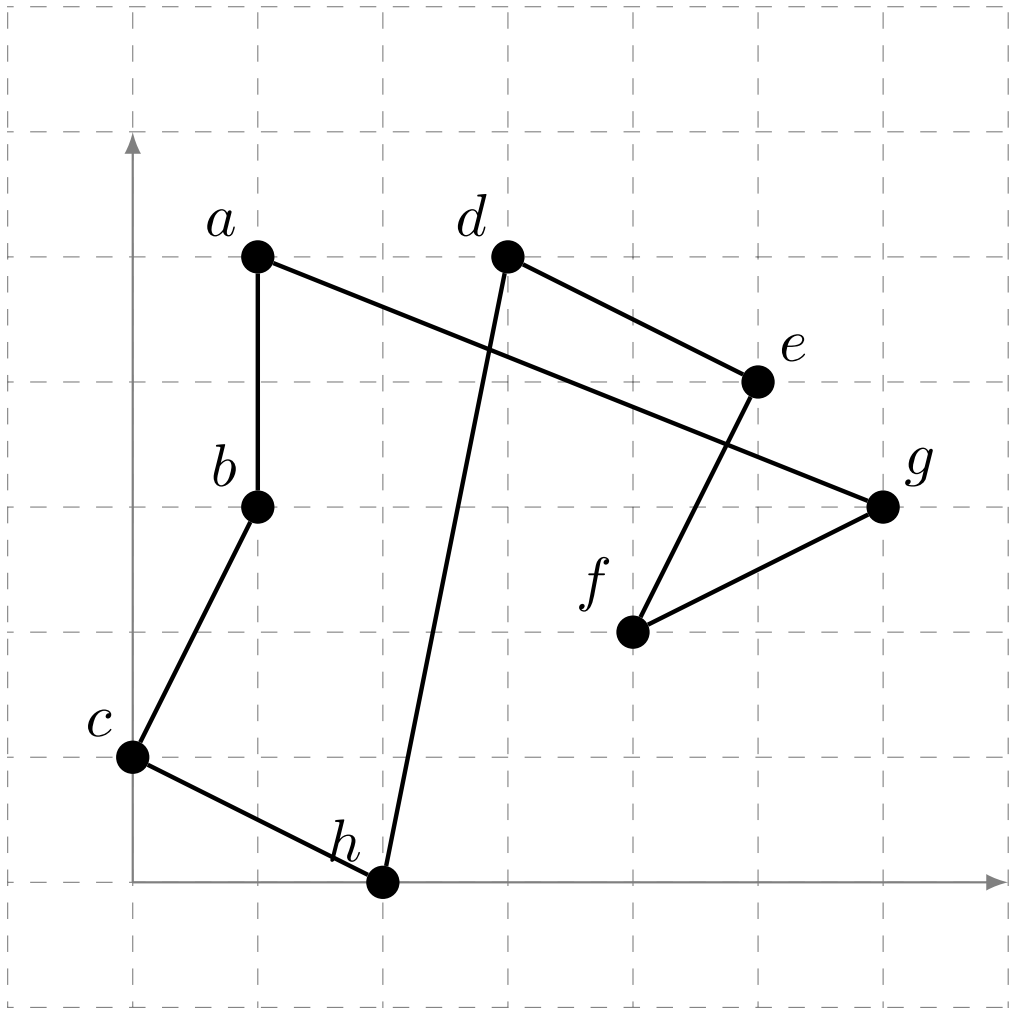
Benaderingsalgoritme: Voorbeeld

Stap 1: Minimale kost opspannende boom.



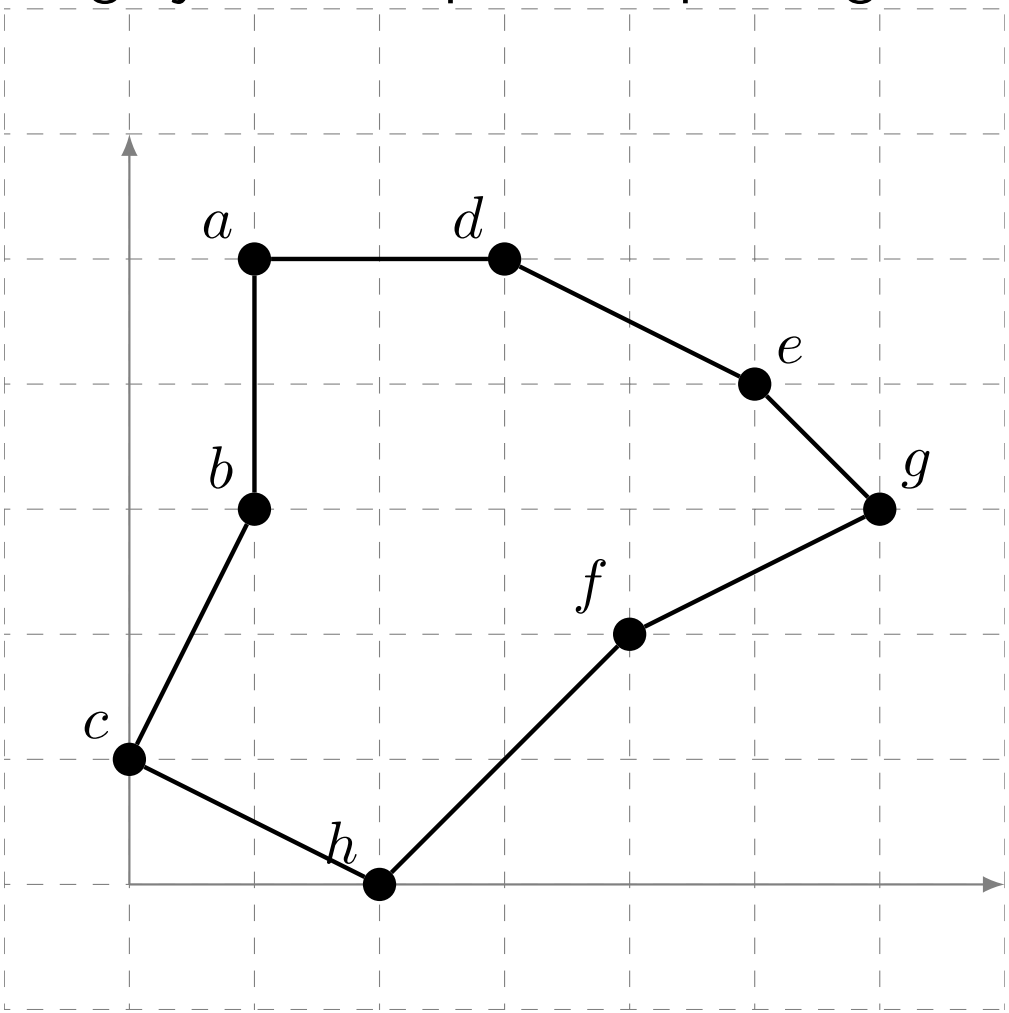
Benaderingsalgoritme: Voorbeeld

Stap 2: Wortel a: preorde doorlopen



Benaderingsalgoritme: Voorbeeld

Vergelijken met optimale oplossing:



Oefening

1. Beschouw opnieuw de acht steden in Figuur 5.18, maar veronderstel nu dat het gewicht van een boog gegeven wordt door de zogenaamde Manhattan-distance tussen de twee knopen, dus

$$d\big((x_1, y_1), (x_2, y_2)\big) = |x_2 - x_1| + |y_2 - y_1|$$

   1.1 Ga na dat de Manhattan-distance aan de driehoeksongelijkheid voldoet. (**Hint**: voor de absolute waarde geldt dat $|x + y| \leq |x| + |y|$.)

   1.2 Pas het benaderende algoritme voor het oplossen van het handelsreizigersprobleem toe op dit probleem. Gebruik Kruskals algoritme om de minimale opspannende boom te construeren. Wanneer meerdere bogen kunnen gekozen worden, kies dan steeds de lexicografisch kleinste boog. Neem de knoop a als wortel van de opspannende boom.

De complexiteitstheorie classificeert problemen volgens hun *moeilijkheidsgraad*.

De complexiteitstheorie kan zeer wiskundig worden benaderd. Wij geven slechts een informele inleiding.

De complexiteitsklasse P

Definitie

De KLASSE $\mathbf{P}$ *is de verzameling van alle problemen die in polynomiale tijd kunnen opgelost worden door een (deterministisch) algoritme.*

Opmerking:

- Polynomiale tijd betekent dat de uitvoeringstijd van het algoritme $O(n^k)$ is met n de grootte van de invoer en k een constante is (onafhankelijk van n).
- De klasse $\mathbf{P}$ is dus de klasse van de *gemakkelijke* problemen.

Voorbeelden uit de klasse P

- Kortste pad in graaf met positieve gewichten.
- Minimale kost opspannende boom.
- Topologisch sorteren.
- Sorteren van een array van objecten.
- ...

Turing's stopprobleem

Turing's stopprobleem:

> *Schrijf een algoritme A (programma) dat bepaalt of een willekeurig programma P met als input I stopt of niet.*

Men kan bewijzen dat het programma A *niet kan geschreven worden*! Turing's stopprobleem is ONBESLISBAAR.

Stelling

Turing's stopprobleem is onbeslisbaar.

Schets van bewijs

Stel dat A wel bestaat.

$$A(P, I) = \begin{cases} 1 & \text{als programma } P \text{ stopt voor invoer } I \\ 0 & \text{als programma } P \text{ niet stopt voor invoer } I. \end{cases}$$

Dan construeren we programma Q:

$$Q(P) = \begin{cases} \text{stopt} & \text{als } A(P, P) = 0 \\ \text{stopt niet} & \text{als } A(P, P) = 1. \end{cases}$$

En nu voeren we Q uit met als invoer zichzelf:

$$Q(Q) = \begin{cases} \text{stopt} & \text{als } A(Q, Q) = 0, \\ & \text{i.e. programma } Q \text{ stopt niet met zichzelf als invoer} \\ \text{stopt niet} & \text{als } A(Q, Q) = 1, \\ & \text{i.e. programma } Q \text{ stopt wel met zichzelf als invoer.} \end{cases}$$

Beide gevallen leveren een contradictie, dus A bestaat niet.

Voorbeeld reducties

- Om de mediaan te berekenen kan men de rij sorteren. Bepalen van de mediaan *reduceert tot* het sorteren van de rij.
- Om de korste afstand te vinden tussen *alle paren* van knopen kunnen we het algoritme van Dijkstra n keer aanroepen. Het vinden van de korste afstand tussen alle paren van knopen *reduceert tot* het n keer aanroepen van het algoritme van Dijkstra.

Definitie reductie

Definitie

Een probleem π_1 reduceert tot een probleem π_2 wanneer een polynomiaal algoritme voor π_2 kan gebruikt worden om probleem π_1 op te lossen in polynomiale tijd.

Dit betekent dat

$$\pi_2 \in \mathbf{P} \implies \pi_1 \in \mathbf{P}$$

of ook

$$\pi_1 \notin \mathbf{P} \implies \pi_2 \notin \mathbf{P}.$$

In woorden: **als π_1 reduceert tot π_2 dan is π_2 minstens zo moeilijk als π_1**

Compleetheid

Een probleem is COMPLEET voor een klasse van problemen als het tot die klasse behoort en minstens zo moeilijk is als alle problemen uit die klasse.

Definitie

Als C een klasse van problemen is dan is een probleem π C-COMPLEET als en slechts als π tot de klasse C behoort en alle problemen uit de klasse C reduceren naar π.

De klasse NP

We wensen aan te tonen dat het handelsreizigersprobleem "moeilijk" is.
Het handelsreizigersprobleem kan opgelost worden met een exponentieel algoritme, i.e. door in essentie alle mogelijkheden te proberen.
Informeel beschrijft de klasse **NP** alle problemen die kunnen opgelost worden m.b.v. een exponentieel algoritme.
De klasse wordt echter gedefinieerd in termen van efficiëntie *verificatie*.

Definitie

De KLASSE **NP** *bestaat uit de problemen waarvoor oplossingen een lengte hebben die hoogstens polynomiaal is in de lengte van de invoer en waarvoor de correctheid van een oplossing kan geverifieerd worden in polynomiale tijd.*

NP-compleetheid

Alles wat nodig is om tot de klasse $\mathbf{NP}$ te behoren is dat men op een efficiënte manier oplossingen kan herkennen en dit betekent dat de klasse $\mathbf{NP}$ enorm veel problemen omvat.

Wanneer een probleem $\mathbf{NP}$-$\textsc{compleet}$ is dan betekent dit (door definitie van compleetheid) dat dit probleem minstens zo moeilijk is als alle problemen in $\mathbf{NP}$.

Cruciale vraag: bestaan er wel $\mathbf{NP}$-complete problemen?

Bestaan van NP-complete problemen

Cook (1971) en Levin (1973) toonden onafhankelijk van elkaar aan dat er **NP**-complete problemen bestaan.

Karp (1972) gaf een lijst van 21 **NP**-complete problemen.

Sindsdien is bewezen van honderden problemen dat ze **NP**-compleet zijn. Onder andere ook het handelsreizigersprobleem en het knapzakprobleem zijn **NP**-compleet.

De klasse P versus NP

Het is duidelijk dat

$$\mathbf{P} \subseteq \mathbf{NP}.$$

Op dit moment weet niemand wat de relatie is tussen $\mathbf{P}$ en $\mathbf{NP}$.

Oplossing is 1 miljoen dollar waard!

Praktisch betekent een $\mathbf{NP}$-compleet probleem dat we zullen moeten werken met benaderende algoritmen om willekeurig grote instanties op te lossen.